The following paragraph does not apply to the United Kingdom or any country or state where such provisions are inconsistent with local law.

The specifications in this manual are subject to change without notice. This manual is provided "AS IS". International Business Machines Corp. makes no warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

International Business Machines Corp. does not warrant that the contents of this publication or the accompanying source code examples, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying source code examples are error-free.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

Address comments to IBM Corporation, 11400 Burnett Road, Austin, Texas 78758-3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

IBM®
Power ISA
PowerPC®
Power Architecture®
PowerPC Architecture
RISC/System 6000
POWER®
POWER2
POWER4
POWER5
POWER7®
POWER8®
POWER9®
System/370

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.
Preface

The roots of the Power ISA (Instruction Set Architecture) extend back 30 years, to IBM Research. The POWER (Performance Optimization With Enhanced RISC) Architecture was introduced with the RISC System/6000 product family in early 1990. In 1991, Apple, IBM, and Motorola began the collaboration to evolve to the PowerPC Architecture, expanding the architecture’s applicability. In 1997, Motorola and IBM began another collaboration, focused on optimizing PowerPC for embedded systems, which produced Book E.

In 2006, Freescale and IBM collaborated on the creation of the Power ISA Version 2.03, which represented the reunification of the architecture by combining Book E content with the more general purpose PowerPC Version 2.02. The resulting architecture included environment-specific privileged architecture optimizations (two Book IIIs) and optional application-specific facilities (categories) as extensions to a pervasive base architecture.

In support of the OpenPOWER Foundation’s standardization of server architecture, Power ISA Version 3.0 streamlined this integration by choosing a single Book III and a set of widely used categories to become part of the base architecture for all forward-looking Power implementations. All other optional architecture categories were eliminated to ensure increased application portability between Power processors. Legacy embedded applications that require the eliminated material will continue to use V. 2.07B.

Power ISA Version 3.0C took the first step in reintroducing optionality into the architecture as the Power ISA moves to an “open” model governed by the OpenPOWER Foundation. Material later in the preface identifies compliancy subsets of the architecture and the optional features which they comprise.

The Power ISA Version 3.1 consists of three books and a set of appendices.

Book I, Power ISA User Instruction Set Architecture, covers the base instruction set and related facilities available to the application programmer.

Book II, Power ISA Virtual Environment Architecture, defines the storage model and other instructions and facilities that enable the application programmer to create multithreaded programs and programs that interact with certain physical realities of the computing environment.

Book III, Power ISA Operating Environment Architecture, defines the supervisor instructions and related facilities.

As used in this document, the term “Power ISA” refers to the instructions and facilities described in Books I, II, and III.

Change bars have been included in the body of this document to indicate changes from the Power ISA Version 3.0C.
Summary of Changes in Power ISA Version 3.1

This document is Version 3.1 of the Power ISA. It is intended to supersede and replace version 3.0C. Any product descriptions that reference a version of the architecture are understood to reference the latest version. This version was created by making miscellaneous corrections and by applying the following requests for change (RFCs) to Power ISA Version 3.0C. Change bars in this summary of changes indicate changes relative to v3.0C.

**Byte-Reverse Instructions:**
Added new GPR-based byte-reverse instructions.

**Vector Integer Multiply/Divide/Modulo Instructions:**
Added SIMD-equivalent forms of FXU multiply, divide, and modulo instructions to increase synergy with FXU instruction set for auto-vectorization.

**Instruction Prefix Support:**
Added a 32-bit instruction prefix to support PC-relative addressing, up to 34-bit immediate operands, additional operand fields, and additional opcode space.

**BHRB Filtering:**
Added new BHRB Filtering fields and defined associated terminology.

**VSX 32-byte Storage Access Operations:**
Added new 32-byte VSR load and store instructions.

**Multiple DEAW:**
Added a second Data Address Watchpoint. [H]DAR is set to the first byte of overlap. 512B boundary is removed. Match detection is on DW granularity independent of operand size. SIAR/SDAR are not altered by the Trace interrupt when TE=0b00.

**128-bit Binary Integer Operations:**
Added new 128-bit integer instructions for comparison, divide, modulo, rotate, shift, DFP and QFP format conversion operations. Also added 128-bit integer multiply assist operations.

**SIMD Permute-Class Operations:**
New permute-class instructions for element extraction and insertion operations, 32-bit immediate splat operations, doublewide bit shift left/right operations, element mask-based blend operations, and an arbitrary-wide permute assist operation.

**Reduced-Precision: Outer Product Operations:**
Added new outer-product instructions to accelerate matrix multiplication, supporting 4-bit, 8-bit, and 16-bit integer and 16-bit, 32-bit, and 64-bit floating-point data types.

**Bit-Manipulation Operations:**
Added new bit-manipulation instructions.

**Set Boolean Extension:**
Added four new instructions that convert a condition code bit (any CR bit) into a Boolean (0/1), the negation of a Boolean (1/0), a field mask (all 0s/all 1s), and the negation of a field mask (all 1s/all 0s) that is placed into a GPR.

**String Operations:**
Added new string isolate instructions to support null-terminated and explicit-length strings.

**Test LSB by Byte Operation:**
Added new instruction to set any CR field to reflect predicate compare summary status, not just CR field 6 which Rc=1 is limited to.

**VSX Load/Store Rightmost Element Operations:**
Added new load and store instructions that transfer the rightmost vector element between VSR and storage.

**Prefixed addi Instruction and Prefixed Load/Store Instructions and Addressing:**
Using new instruction prefix, added support for extended immediate displacements and PC-relative addressing for a specific set of GPR and VSR load and store operations.

**VSX Scalar Minimum/Maximum/Compare Quad-Precision Operations:**
Add new quad-precision minimum, maximum, and predicate comparison instructions.

**CMODX Extension for Prefix:**
The quasi patch class of unsynchronized updates to instruction storage is made architecture. Language is changed and rules are added to account for the addition of prefixed instructions to the architecture.

**Reduced-Precision - bfloat16 Outer Product & Format Conversion Operations:**
Added new instructions to accelerate matrix multiplication and format conversions for the bfloat16 datatype.

**Processor Control Register Extensions:**
The PCR is updated to accommodate new problem-state instructions added in v3.1.

**Reduced-Precision: Missing Integer-based Outer Product Operations:**
Added additional new instructions to accelerate matrix multiplication for 8-bit and 16-bit integer datatypes.

**VSX Mask Manipulation Operations:**
Added new vector instructions to manipulate vector masks.

**VSX PCV Generate Operations:**
Added new permute control vector generate instructions to support efficient emulation of load expand and store compress operations.
New Performance Monitor SPRs:
Added three new performance monitor SPRs. SIER2 and SIER3 are added to provide additional information about the sampled instruction. MMCR3 is added for further sampling related configuration control.

Translation Management Extensions:
Added an L bit for sliag, where L=1 indicates an invalidation by LPID. tbiel with SET=0 and IS=1, 2, or 3 invalidate all congruence classes and tbiel with SET !=0 is a noop except when RIC=1, which becomes an invalid form. Made ISL apply in hypervisor state.

Copy/Paste Extensions:
Added memory move functionality.

Persistent Storage / Store Sync:
Added pushes and synchronization for persistent storage and variants of sync optimized for store ordering.

Pause / Wait-reserve:
Added two new variants of the wait instruction; removed platform notify, TIDR, and CIR.

Performance Monitor Facility Sampling Security:
Changes the definition of MMCR0_{PMCC=0b00} case to allow for a new secure mode of access with regards to sampling registers which is available conditional on new MMCR_{PMCCEXT} bit. Introduces a new freeze mode for ultravisor privilege state differentiating it from hypervisor privilege state freeze mode. Restricts BHRB to only record in problem state. Also MMCR0_{PMAQ}, bit 52 of MMCR0 is removed.

Hypervisor Interrupt Location Control:
Added HAIL for the hypervisor to specify its interrupt behavior independent from guest state.

Changes and Clarifications to Data Cache Management Instructions:
Specifies the that the number of software data prefetch streams guaranteed to be available to a thread varies by degree of multithreading in the processor. Clarifies when a new software data prefetch stream will overwrite an existing one. Redefines when a thread’s software data prefetch streams are cleared.

BHRB Disable Control:
Adds an additional control on BHRB recording via MMCRA bit 26 namely MMCRA_{BHRBRD}.
OpenISA Compliancy Subset Methodology and Requirements

The PowerISA comprises the base architecture (that which is never optional - not part of any optional or deprecated feature), four groups of optional features, and a group of deprecated features. (See the next two pages.) Authorized implementations of the Power ISA must support one of the four Compliancy Subsets defined below. Support of a subset means that a design includes the base architecture and all features that are not optional for that subset. A supporting design may also include any features that are optional for the supported subset (including deprecated features), subject to stated pre-requisites, and Custom Extensions created using the architecture sandbox defined below. “Inclusion” of the base architecture, of an optional feature, or of a custom extension can be accomplished using a combination of hardware and firmware, provided that the firmware is implemented using other elements of the base architecture and of the included features, and elements of the architecture sandbox; invoked using the second and third pages of real storage (see the second bullet of Section 6.7.5 of Book III); and subject to the prohibitions against the use of firmware given in Section 7.4.4 of Book III. Each optional or deprecated feature must be implemented in its entirety. Attempted execution of an instruction associated with a feature that is not included must cause a Hypervisor Emulation Assistance Interrupt (HEAI). The result of an attempted access to an SPR associated with a feature that is not included using mtspr or mfspr must be that described for “an SPR number that is undefined for the implementation” in the respective instruction description. See Section 5.4.4 of Book III. For Scalar Fixed-Point + Floating-Point and Scalar Fixed-Point Compliancy Subset implementations that do not include the logical partitioning feature, an Illegal Instruction type Program Interrupt as described in the penultimate Programming Note in Section 7.5.9 of Book III may be substituted for the HEAI.

OpenPOWER Compliancy Subsets

1. **AIX Compliancy Subset (ACS)**
   - The following features are optional for this compliancy subset. The rest of PowerISA v3.1 must be included.
     - Always Optional Features listed below
     - Deprecated Features listed below

2. **Linux Compliancy Subset (LCS)**
   - The following features are optional for this compliancy subset. The rest of PowerISA v3.1 must be included.
     - Linux Optional Features listed below
     - Always Optional Features listed below
     - Deprecated Features listed below

3. **Scalar Fixed-Point + Floating-Point Compliancy Subset (SFFS)**
   - The following features are optional for this compliancy subset. The rest of PowerISA v3.1 must be included.
     - Scalar Float Optional Features listed below
     - Linux Optional Features listed below
     - Always Optional Features listed below
     - Deprecated Features listed below

4. **Scalar Fixed-Point Compliancy Subset (SFS)**
   - The following features are optional for this compliancy subset. The rest of PowerISA v3.1 must be included.
     - Scalar Fixed Optional Features listed below
     - Scalar Float Optional Features listed below
     - Linux Optional Features listed below
     - Always Optional Features listed below
     - Deprecated Features listed below
OpenPOWER Optional and Deprecated Features

Always Optional Features

The following features are optional for all compliancy subsets.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy/Paste for accelerator invocation and memory copy (CPA)</td>
<td>See Section 4.4 of Book II.</td>
</tr>
<tr>
<td>Secure Memory Facility (SMF)¹</td>
<td>See Chapter 3 of Book III.</td>
</tr>
<tr>
<td>Hardware and software data stream prefetching (STM) (DSCR state not optional)</td>
<td>See Section 4.2 and Section 4.3.2 of Book II.</td>
</tr>
<tr>
<td>M=0 (M) (non-coherent memory)</td>
<td>See Section 1.6.3 of Book II.</td>
</tr>
<tr>
<td>W=1 (W) (write through-required memory)</td>
<td>See Section 1.6.1 of Book II.</td>
</tr>
<tr>
<td>Power management (PM)²</td>
<td>See Section 4.2.2, Section 4.3.2, and the description of the PECE field(s) of the LPCR in Section 2.2 of Book III.</td>
</tr>
<tr>
<td>MMA³</td>
<td>See Section 7.2.1.3 and Section 7.6.1.12 of Book I.</td>
</tr>
</tbody>
</table>

Notes:

1. LPAR is a pre-requisite for SMF.
2. If Power management is implemented by an ACS- or LCS-compliant design, it must be implemented as the architecture describes. If Power management is implemented by an SFFS- or SFS-compliant design, it need not be implemented as the architecture describes, and may include different interfaces created from the architecture sandbox.
3. SIMD is a requirement for MMA.
Linux Optional Features

The following features are optional for the Linux Compliancy Subset, the Scalar Fixed-Point + Floating-Point Compliancy Subset, and the Scalar Fixed-Point Compliancy Subset.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIL/HAIL programmability (AIL) (AIL=3 and HAIL=1 required)</td>
<td>See the description of the AIL and HAIL fields of the LPCR in Section 2.2 of Book III.</td>
</tr>
<tr>
<td>Atomic Memory Operations (AMO)</td>
<td>See Section 4.5 of Book II.</td>
</tr>
<tr>
<td>Big Endian (BE) (LE is required for LCS. Linux supporting LCS is 64b LE Linux.)</td>
<td>See Section 1.10 of Book I and its first two subsections. Also see the description of the ILE field of the LPCR in Section 2.2 of Book III and the description of the LE bit of the MSR in Section 4.2.1 of Book III.</td>
</tr>
<tr>
<td>Branch History Rolling Buffer (BHRB)</td>
<td>See Chapter 7 of Book II.</td>
</tr>
<tr>
<td>Decimal floating-point (DFP)¹</td>
<td>See Chapter 5 of Book I.</td>
</tr>
<tr>
<td>Event-Based Branching (EBB)</td>
<td>See Chapter 6 of Book II.</td>
</tr>
<tr>
<td>EVlRT programmability (EVlRT)² (EVlRT=1 required)</td>
<td>See the description of the EVlRT field of the LPCR in Section 2.2 of Book III.</td>
</tr>
<tr>
<td>SLB / HPT translation (HPT) (includes VPM, ISL, KBV)</td>
<td>See Section 6.7.7 through Section 6.7.9 of Book III. Also see the description of the VPM, ISL, and KBV fields of the LPCR in Section 2.2 of Book III.</td>
</tr>
<tr>
<td>Load/Store Multiple instructions (LM)</td>
<td>See Section 3.3.6 of Book I.</td>
</tr>
<tr>
<td>Load/Store String instructions (LS)</td>
<td>See Section 3.3.7 of Book I.</td>
</tr>
<tr>
<td>Processor Compatibility Register (PCR)²</td>
<td>See Section 2.5 of Book III.</td>
</tr>
<tr>
<td>Quad-precision floating-point (QFP)³</td>
<td>See Chapter 7 of Book I.</td>
</tr>
<tr>
<td>Broadcast TLB shootdown (TLBIE) (tlbiel not optional)</td>
<td>See Section 6.9.3.3 of Book III.</td>
</tr>
<tr>
<td>Control Register (CTRL)</td>
<td>See Section 5.3.4 of Book III.</td>
</tr>
<tr>
<td>SMT (SMT)⁴ (includes PURR/SPURR, PSPB, RPR, PPR, processor control) (PPR and hypervisor/ultrvisor messaging not optional)</td>
<td>See Chapter 3 of Book II. Also see Section 5.3.5 through Section 5.3.7, Section 8.6, Section 8.7, and Chapter 11 of Book III.</td>
</tr>
</tbody>
</table>

Notes:

1. FP is a pre-requisite for DFP.
2. LPAR is a pre-requisite for EVlRT and PCR.
3. SIMD is a pre-requisite for QFP.
4. If SMT is implemented by an LCS-compliant design, it must be implemented as the architecture describes. If SMT is not implemented by an LCS-compliant design, the design must not except on PPR accesses and must implement msgsnd[], msgctl[], and msgsyc. If SMT is implemented by an SFFS- or SFS-compliant design, it need not be implemented as the architecture describes, and may include different interfaces created from the architecture sandbox.
Scalar Float Optional Features

The following features are optional for the Scalar Fixed-Point + Floating-Point Compliancy Subset and the Scalar Fixed-Point Compliancy Subset.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD (SIMD)(^1) (VMX and VSX)</td>
<td>See Chapter 6 and Chapter 7 of Book I.</td>
</tr>
<tr>
<td>SF=1 (64-bit)(^2)</td>
<td>See Section 1.5 and Section 1.10.3 of Book I and the description of the SF field of the MSR in Section 4.2.1 of Book III.</td>
</tr>
<tr>
<td>Little Endian (LE) (BE is required for SFFS and SFS. Linux supporting SFFS and SFS is 32b BE Linux.)</td>
<td>See Section 1.10 of Book I and its first two subsections. Also see the description of the ILE field of the LPCR in Section 2.2 of Book III and the description of the LE bit of the MSR in Section 4.2.1 of Book III.</td>
</tr>
<tr>
<td>Logical partitioning (LPAR)(^3)(^4)</td>
<td>See Chapter 2 of Book III.</td>
</tr>
<tr>
<td>Fixed-point instructions that modify OV to indicate whether overflow occurred (OV) (addex and instructions with OE=1 such as addo, subfo, etc.)</td>
<td>See Section 3.3.9 of Book I.</td>
</tr>
<tr>
<td>Nested radix translation (ROR)(^5) (single-level radix translation not optional)</td>
<td>See Section 6.7.7 and Section 6.7.10 of Book III.</td>
</tr>
</tbody>
</table>

Notes:
1. FP is a pre-requisite for SIMD.
2. When 64-bit is not included, a single radix tree will be used to map both application and OS address spaces (no quadrant structure).
3. 64-bit is a pre-requisite for LPAR.
4. When LPAR is not included, MSRHV=1 always.
5. LPAR is a pre-requisite for ROR.

Scalar Fixed Optional Features

The following features are optional for the Scalar Fixed-Point Compliancy Subset

<table>
<thead>
<tr>
<th>Feature</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar binary floating-point (FP)</td>
<td>See Chapter 4 of Book I.</td>
</tr>
</tbody>
</table>

Deprecated Features

There are no deprecated features in Power ISA v3.1.
OpenPOWER Architecture Sandbox

OpenPOWER compliance subsets permit Custom Extensions. Any architectural resources used for Custom Extensions must use only the resources described below and any instructions and SPRs that the architecture describes as implementation-dependent.

Development of Custom Extensions using the architecture sandbox is appropriate for facilities that benefit a small portion of the processor design space. For facilities with broad applicability, developers are strongly encouraged to submit a proposal for adoption into the architecture. Adopted proposals will become optional or required features of the architecture, and will be assigned resources that are not in the architecture sandbox to avoid fragmentation of the architecture. Facilities described in proposals that are not adopted into the architecture may be implemented as Custom Extensions using the architecture sandbox.

System software and toolchain support of Custom Extensions is not guaranteed. Developers are encouraged to provide a means to disable custom extensions to present an architecture that is supported by standard system software and toolchain.

The architecture sandbox consists of the following.
- The designated opcode sandbox is instructions having a primary opcode of 22. Note that primary opcode 22 is reserved by AIX. As a result, Custom Extensions that use primary opcode 22 are not compatible with ACS.
- The designated SPR sandbox consists of non-privileged SPRs 704-719 and privileged SPRs 720-735.
- The designated [H]FSCR sandbox consists of [H]FSCR bits 8-9 and their corresponding IC values.
- The designated XER bit sandbox consists of XER bits 54:55.
- The designated FPSCR bit sandbox consists of FPSCR bits 14-15.
- The designated VSCR bit sandbox consists of VSCR bits 96 & 112. VSCR bit 96 is provided for Vector Facility control & VSCR bit 112 is provided for Vector Facility status.
- The designated interrupt vector sandbox consists of interrupt vector 0x0000_0000_0000_0FE0.
Table of Contents

Preface ................................................................ v
  Summary of Changes in Power ISA Version 3.1 .......... vi
  OpenISA Compliancy Subset Methodology and Requirements ... viii

Table of Contents .............................................. xiii

Book I: Power ISA User Instruction Set Architecture .......... 1

Chapter 1. Introduction .................................. 3
  1.1 Overview ............................................. 3
  1.2 Instruction Mnemonics and Operands ................. 3
  1.3 Document Conventions ............................... 3
    1.3.1 Definitions ..................................... 3
    1.3.2 Notation ....................................... 4
    1.3.3 Reserved Fields, Reserved Values, and Reserved SPRs ... 5
    1.3.4 Description of Instruction Operation ........... 6
    1.3.5 Phased-Out Facilities ......................... 8
    1.4 Processor Overview ................................ 9
    1.5 Computation modes ................................ 10
    1.6 Instruction Formats ................................ 11
    1.6.1 Word Instruction Formats ...................... 12
    1.6.1.1 A-FORM ..................................... 12
    1.6.1.2 B-FORM ..................................... 12
    1.6.1.3 D-FORM ..................................... 12
    1.6.1.4 DQ-FORM .................................... 12
    1.6.1.5 DS-FORM .................................... 12
    1.6.1.6 DX-FORM .................................... 12
    1.6.1.7 I-FORM ...................................... 12
    1.6.1.8 M-FORM ...................................... 12
    1.6.1.9 MD-FORM .................................... 12
    1.6.1.10 MDS-FORM ................................ 12
    1.6.1.11 SC-FORM .................................... 12
    1.6.1.12 VA-FORM .................................... 12
    1.6.1.13 VC-FORM .................................... 12
    1.6.1.14 VX-FORM .................................... 13
    1.6.1.15 X-FORM ..................................... 13
    1.6.1.16 XFL-FORM .................................. 15
    1.6.1.17 XFX-FORM .................................. 15
    1.6.1.18 XL-FORM .................................... 15
    1.6.1.19 XO-FORM .................................... 15
    1.6.1.20 XS-FORM .................................... 15
    1.6.1.21 XX2-FORM .................................. 15
    1.6.1.22 XX3-FORM .................................. 15
    1.6.1.23 XX4-FORM .................................. 15
    1.6.1.24 Z22-FORM .................................. 15
    1.6.1.25 Z23-FORM .................................. 16
    1.6.2 Word Instruction Fields .......................... 16
    1.6.3 Instruction Prefix Formats ....................... 22
      1.6.3.1 Type 00 Prefix – Eight-Byte Load/Store Instructions ... 22
      1.6.3.2 Type 01 Prefix – Eight-Byte Register-to-Register Instructions ... 23
    1.6.3.3 Type 10 - Modified Load/Store Instructions ........ 23
      1.6.3.4 Type 11 - Modified Register-to-Register Instructions ... 23
    1.6.4 Instruction Prefix Fields ......................... 23
    1.7 Classes of Instructions ............................. 24
      1.7.1 Defined Instruction Class ....................... 24
      1.7.2 Illegal Instruction Class ....................... 24
      1.7.3 Reserved Instruction Class ...................... 24
    1.8 Forms of Defined Instructions ....................... 24
      1.8.1 Preferred Instruction Forms .................... 24
      1.8.2 Invalid Instruction Forms ....................... 24
      1.8.3 Reserved-no-op Instructions ..................... 25
    1.9 Exceptions .......................................... 25
      1.9.1 Storage Addressing ............................. 25
      1.9.2 Storage Operands ................................ 26
      1.9.3 Instruction Fetches .............................. 28
      1.9.4 Effective Address Calculation .................... 29

Chapter 2. Branch Facility ............................. 33
  2.1 Branch Facility Overview .......................... 33
  2.2 Instruction Execution Order ......................... 33
  2.3 Branch Facility Registers ........................... 34
    2.3.1 Condition Register ............................. 34
    2.3.2 Link Register .................................. 35
    2.3.3 Count Register ................................. 35
    2.3.4 Target Address Register ......................... 35
  2.4 Branch Instructions ................................ 37
  2.5 Condition Register Instructions ...................... 44
Chapter 3. Fixed-Point Facility ... 49

3.1 Fixed-Point Facility Overview ..... 49
3.2 Fixed-Point Facility Registers ..... 49
3.2.1 General Purpose Registers ..... 49
3.2.2 Fixed-Point Exception Register ..... 49
3.2.3 VR Save Register ..... 50
3.3 Fixed-Point Facility Instructions ..... 51
3.3.1 Fixed-Point Storage Access Instructions ..... 51
3.3.1.1 Storage Access Exceptions ..... 51
3.3.1.2 Fixed-Point Storage Access Instructions ..... 56
3.3.2 Fixed-Point Load Instructions ..... 51
3.3.2.1 64-bit Fixed-Point Load Instructions ..... 56
3.3.3 Fixed-Point Store Instructions ..... 58
3.3.3.1 64-bit Fixed-Point Store Instructions ..... 62
3.3.4 Fixed Point Load and Store Quad-word Instructions ..... 64
3.3.5 Fixed-Point Load and Store with Byte Reversal Instructions ..... 67
3.3.5.1 64-Bit Load and Store with Byte Reversal Instructions ..... 69
3.3.6 Fixed-Point Load and Store Multiple Instructions ..... 70
3.3.7 Fixed-Point Move Assist Instructions [Phased Out] ..... 71
3.3.8 Other Fixed-Point Instructions ..... 74
3.3.9 Fixed-Point Arithmetic Instructions 75
3.3.9.1 64-bit Fixed-Point Arithmetic Instructions ..... 87
3.3.10 Fixed-Point Compare Instructions ..... 92
3.3.10.1 Character-Type Compare Instructions ..... 94
3.3.11 Fixed-Point Trap Instructions ..... 96
3.3.11.1 64-bit Fixed-Point Trap Instructions ..... 98
3.3.12 Fixed-Point Select ..... 98
3.3.13 Fixed-Point Logical Instructions 99
3.3.13.1 64-bit Fixed-Point Logical Instructions 100
3.3.14 Fixed-Point Rotate and Shift Instructions 107
3.3.14.1 Fixed-Point Rotate Instructions 107
3.3.14.1.1 64-bit Fixed-Point Rotate Instructions 110
3.3.14.2 Fixed-Point Shift Instructions 113
3.3.14.2.1 64-bit Fixed-Point Shift Instructions 115
3.3.15 Binary Coded Decimal (BCD) Assist Instructions 117
3.3.16 Byte-Reverse Instructions 119
3.3.17 Move To/From Vector-Scalar Register Instructions 120
3.3.18 Move To/From System Register Instructions 124
3.3.19 Prefixed No-Operation Instruction 130

Chapter 4. Floating-Point Facility 131

4.1 Floating-Point Facility Overview 131
4.2 Floating-Point Facility Registers 132
4.2.1 Floating-Point Registers 132
4.2.2 Floating-Point Status and Control Register 132
4.3 Floating-Point Data 135
4.3.1 Data Format 135
4.3.2 Value Representation 135
4.3.3 Sign of Result 137
4.3.4 Normalization and Denormalization 137
4.3.5 Data Handling and Precision 137
4.3.5.1 Single-Precision Operands 137
4.3.5.2 Integer-Valued Operands 138
4.3.6 Rounding 139
4.4 Floating-Point Exceptions 140
4.4.1 Invalid Operation Exception 142
4.4.1.1 Definition 142
4.4.1.2 Action 142
4.4.2 Zero Divide Exception 142
4.4.2.1 Definition 142
4.4.2.2 Action 143
4.4.3 Overflow Exception 143
4.4.3.1 Definition 143
4.4.3.2 Action 143
4.4.4 Underflow Exception 144
4.4.4.1 Definition 144
4.4.4.2 Action 144
4.4.5 Inexact Exception 144
4.4.5.1 Definition 144
4.4.5.2 Action 144
4.5 Floating-Point Execution Models 145
4.5.1 Execution Model for IEEE Operations 145
4.5.2 Execution Model for Multiply-Add Type Instructions 147
4.6 Floating-Point Facility Instructions 148
4.6.1 Floating-Point Storage Access Instructions 148
4.6.1.1 Storage Access Exceptions 148
4.6.2 Floating-Point Load Instructions 149
4.6.3 Floating-Point Store Instructions 154
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.8.6 Vector Select Instruction</td>
<td>297</td>
</tr>
<tr>
<td>6.8.7 Vector Shift Instructions</td>
<td>298</td>
</tr>
<tr>
<td>6.8.8 Vector Extract Element Instructions</td>
<td>304</td>
</tr>
<tr>
<td>6.8.8.1 Vector Extract Element to VSR using Immediate-specified Index Instructions</td>
<td>304</td>
</tr>
<tr>
<td>6.8.8.2 Vector Extract Element to GPR using GPR-specified Index Instructions</td>
<td>306</td>
</tr>
<tr>
<td>6.8.8.3 Vector Extract Double Element to VSR Using GPR-specified Index Instructions</td>
<td>309</td>
</tr>
<tr>
<td>6.8.9 Vector Insert Element Instructions</td>
<td>313</td>
</tr>
<tr>
<td>6.8.9.1 Vector Insert Element from VSR Using Immediate-specified Index Instructions</td>
<td>313</td>
</tr>
<tr>
<td>6.8.9.2 Vector Insert Element from GPR Using GPR-specified Index Instructions</td>
<td>315</td>
</tr>
<tr>
<td>6.8.9.3 Vector Insert Element from GPR Using Immediate-specified Index Instructions</td>
<td>319</td>
</tr>
<tr>
<td>6.8.9.4 Vector Insert Element from VSR Using GPR-specified Index Instructions</td>
<td>320</td>
</tr>
<tr>
<td>6.9 Vector Integer Instructions</td>
<td>323</td>
</tr>
<tr>
<td>6.9.1 Vector Integer Arithmetic Instructions</td>
<td>323</td>
</tr>
<tr>
<td>6.9.1.1 Vector Integer Add Instructions</td>
<td>323</td>
</tr>
<tr>
<td>6.9.1.2 Vector Integer Subtract Instructions</td>
<td>331</td>
</tr>
<tr>
<td>6.9.1.3 Vector Integer Multiply Instructions</td>
<td>339</td>
</tr>
<tr>
<td>6.9.1.4 Vector Integer Multiply-Add/Sum Instructions</td>
<td>351</td>
</tr>
<tr>
<td>6.9.1.5 Vector Integer Divide Instructions</td>
<td>358</td>
</tr>
<tr>
<td>6.9.1.6 Vector Integer Modulo Instructions</td>
<td>364</td>
</tr>
<tr>
<td>6.9.1.7 Vector Integer Sum-Across Instructions</td>
<td>367</td>
</tr>
<tr>
<td>6.9.1.8 Vector Integer Negate Instructions</td>
<td>371</td>
</tr>
<tr>
<td>6.9.1.9 Vector Extend Sign Instructions</td>
<td>372</td>
</tr>
<tr>
<td>6.9.1.10 Vector Integer Average Instructions</td>
<td>375</td>
</tr>
<tr>
<td>6.9.1.11 Vector Integer Absolute Difference Instructions</td>
<td>378</td>
</tr>
<tr>
<td>6.9.2 Vector Integer Maximum/Minimum Instructions</td>
<td>380</td>
</tr>
<tr>
<td>6.9.2.1 Vector Integer Maximum Instructions</td>
<td>380</td>
</tr>
<tr>
<td>6.9.2.2 Vector Integer Minimum Instructions</td>
<td>384</td>
</tr>
<tr>
<td>6.9.3 Vector Integer Compare Instructions</td>
<td>388</td>
</tr>
<tr>
<td>6.9.4 Vector Logical Instructions</td>
<td>402</td>
</tr>
<tr>
<td>6.9.5 Vector Integer Rotate Instructions</td>
<td>404</td>
</tr>
<tr>
<td>6.9.5.1 Vector Integer Rotate Left Instructions</td>
<td>404</td>
</tr>
<tr>
<td>6.9.5.2 Vector Integer Rotate Left then AND with Mask Instructions</td>
<td>407</td>
</tr>
<tr>
<td>6.9.5.3 Vector Integer Rotate Left then Mask Insert Instructions</td>
<td>410</td>
</tr>
<tr>
<td>6.9.6 Vector Integer Shift Instructions</td>
<td>413</td>
</tr>
<tr>
<td>6.9.6.1 Vector Integer Shift Left Instructions</td>
<td>413</td>
</tr>
<tr>
<td>6.9.6.2 Vector Integer Shift Right Instructions</td>
<td>416</td>
</tr>
<tr>
<td>6.9.6.3 Vector Integer Shift Right Algebraic Instructions</td>
<td>419</td>
</tr>
<tr>
<td>6.10 Vector Floating-Point Instruction Set</td>
<td>422</td>
</tr>
<tr>
<td>6.10.1 Vector Floating-Point Arithmetic Instructions</td>
<td>422</td>
</tr>
<tr>
<td>6.10.2 Vector Floating-Point Maximum/Minimum Instructions</td>
<td>424</td>
</tr>
<tr>
<td>6.10.3 Vector Floating-Point Rounding and Conversion Instructions</td>
<td>425</td>
</tr>
<tr>
<td>6.10.3.1 Vector Floating-Point Conversion Instructions</td>
<td>425</td>
</tr>
<tr>
<td>6.10.3.2 Vector Floating-Point Round to Integral Instructions</td>
<td>427</td>
</tr>
<tr>
<td>6.10.4 Vector Floating-Point Compare Instructions</td>
<td>429</td>
</tr>
<tr>
<td>6.10.5 Vector Floating-Point Estimate Instructions</td>
<td>429</td>
</tr>
<tr>
<td>6.11 Vector Exclusive-OR-based Instructions</td>
<td>432</td>
</tr>
<tr>
<td>6.11.1 Vector AES Instructions</td>
<td>435</td>
</tr>
<tr>
<td>6.11.2 Vector SHA-256 and SHA-512 Sigma Instructions</td>
<td>438</td>
</tr>
<tr>
<td>6.11.3 Vector Binary Polynomial Multiplication Instructions</td>
<td>440</td>
</tr>
<tr>
<td>6.11.4 Vector Permute &amp; Exclusive-OR Instruction</td>
<td>444</td>
</tr>
<tr>
<td>6.12 Vector Bit Manipulation Instructions</td>
<td>445</td>
</tr>
<tr>
<td>6.12.1 Vector Gather Bits Instructions</td>
<td>445</td>
</tr>
<tr>
<td>6.12.2 Vector Count Leading Zeros Instructions</td>
<td>447</td>
</tr>
<tr>
<td>6.12.3 Vector Count Trailing Zeros Instructions</td>
<td>450</td>
</tr>
<tr>
<td>6.12.4 Vector Count Leading/Trailing Zero LSB Instructions</td>
<td>453</td>
</tr>
<tr>
<td>6.12.5 Vector Bit Insert/Extract Instructions</td>
<td>454</td>
</tr>
<tr>
<td>6.12.6 Vector Centrifuge Instruction</td>
<td>456</td>
</tr>
</tbody>
</table>
Chapter 7. Vector-Scalar Extension Facility

7.1 Introduction
7.1.1 Overview of the Vector-Scalar Extension
7.1.1.1 Combining the Floating-Point Registers (FPR) defined in Chapter 4. Floating-Point Facility and the Vector Registers (VR) defined in Chapter 6. Vector Facility provides additional registers to support more aggressive compiler optimizations for both vector and scalar operations. Compatibility with Floating-Point and Decimal Floating-Point Operations
7.1.1.2 Compatibility with Vector Operations
7.2 VSX Registers
7.2.1 Vector-Scalar Registers
7.2.1.1 Floating-Point Registers
7.2.1.2 Vector Registers
7.2.1.3 VSX Accumulators
7.2.2 Floating-Point Status and Control Register
7.3 VSX Operations

7.3.1 VSX Floating-Point Arithmetic Overview
7.3.2 VSX Floating-Point Data
7.3.2.1 Data Format
7.3.2.2 Value Representation
7.3.2.3 Sign of Result
7.3.2.4 Normalization and Denormalization
7.3.2.5 Data Handling and Precision
7.3.2.6 Rounding
7.3.3 VSX Floating-Point Execution Models
7.3.3.1 VSX Execution Model for IEEE Operations
7.3.3.2 VSX Execution Model for Multiply-Add Type Instructions
7.4 VSX Floating-Point Exceptions
7.4.1 Floating-Point Exceptions
7.4.1.1 Definition
7.4.1.2 Action for VE=1
7.4.1.3 Action for VE=0
7.4.2 Floating-Point Zero Divide Exception
7.4.2.1 Definition
7.4.2.2 Action for ZE=1
7.4.2.3 Action for ZE=0
7.4.3 Floating-Point Overflow Exception
7.4.3.1 Definition
7.4.3.2 Action for OE=1
7.4.3.3 Action for OE=0
7.4.4 Floating-Point Underflow Exception
7.4.4.1 Definition
7.4.4.2 Action for UE=1
7.4.4.3 Action for UE=0
7.4.5 Floating-Point Inexact Exception
7.4.5.1 Definition
7.4.5.2 Action for XE=1
7.4.5.3 Action for XE=0
7.5 VSX Storage Access Operations
7.5.1 Accessing Aligned Storage Operations
7.5.2 Accessing Unaligned Storage Operations
7.5.3 Storage Access Exceptions
7.6 VSX Instruction Set
7.6.1 VSX Instruction Set Summary
7.6.1.1 VSX Storage Access Instructions
7.6.1.2 VSX Binary Floating-Point Sign Manipulation Instructions
7.6.1.3 VSX Binary Floating-Point Arithmetic Instructions
7.6.1.4 VSX Binary Floating-Point Compare Instructions .................. 570
7.6.1.5 VSX Binary Floating-Point Round to Shorter Precision Instructions .... 571
7.6.1.6 VSX Binary Floating-Point Convert to Shorter Precision Instructions .... 571
7.6.1.7 VSX Binary Floating-Point Convert to Longer Precision Instructions .... 571
7.6.1.8 VSX Binary Floating-Point Round to Integral Instructions ............. 572
7.6.1.9 VSX Binary Floating-Point Convert to Integer Instructions ............ 572
7.6.1.10 VSX Binary Floating-Point Convert From Integer Instructions ......... 573
7.6.1.11 VSX Binary Floating-Point Math Support Instructions ................ 573
7.6.1.12 VSX Matrix-Multiply Assist (MMA) Instructions ...................... 574
7.6.1.12.1 VSX Accumulator Move Instructions ................................ 574
7.6.1.12.2 VSX Binary Integer Outer-Product Instructions .................... 574
7.6.1.12.3 VSX Binary Floating-Point Outer-Product Instructions .............. 575
7.6.1.13 VSX Vector Logical Instructions ...................................... 576
7.6.1.14 VSX Vector Permute-class Instructions ................................ 577
7.6.1.15 VSX Vector Load Special Value Instruction ............................ 578
7.6.1.16 VSX Vector Test Least-Significant Bit by Byte Instruction ............ 578
7.6.2 VSX Instruction Description Conventions ................................ 579
7.6.2.1 VSX Instruction RTL Operators 579
7.6.2.2 VSX Instruction RTL Function Calls .................................. 580
7.6.3 VSX Instruction Descriptions .............................................. 580

Appendix A. Suggested Floating-Point Models ......................... 995
A.1 Floating-Point Round to Single-Precision Model ...................... 995
A.2 Floating-Point Convert to Integer Model ............................... 999
A.3 Floating-Point Convert from Integer Model ......................... 1002
A.4 Floating-Point Round to Integer Model ............................... 1004

Appendix B. Densely Packed Decimal ................................. 1007
B.1 BCD-to-DPD Translation ................................................. 1007
B.2 DPD-to-BCD Translation ................................................. 1007
B.3 Preferred DPD encoding ................................................ 1008

Appendix C. Assembler Extended Mnemonics ...................... 1011
C.1 Symbols ................................................................. 1011
C.2 Branch Mnemonics ..................................................... 1012
C.2.1 BO and BI Fields .................................................. 1012
C.2.2 Simple Branch Mnemonics ........................................ 1012
C.2.3 Branch Mnemonics Incorporating Conditions ...................... 1013
C.2.4 Branch Prediction .................................................. 1014
C.3 Condition Register Logical Mnemonics ............................ 1015
C.4 Subtract Mnemonics ................................................... 1015
C.4.1 Subtract Immediate ................................................ 1015
C.4.2 Subtract ........................................................... 1015
C.5 Compare Mnemonics .................................................. 1016
C.5.1 Doubleword Comparisons ........................................ 1016
C.5.2 Word Comparisons ................................................ 1016
C.6 Trap Mnemonics ......................................................... 1017
C.7 Integer Select Mnemonics ............................................ 1018
C.8 Rotate and Shift Mnemonics ......................................... 1019
C.8.1 Operations on Doublewords ....................................... 1019
C.8.2 Operations on Words ............................................... 1020
C.9 Move To/From Special Purpose Register Mnemonics ............. 1021
C.10 Miscellaneous Mnemonics ........................................... 1022

Book II:

Power ISA Virtual Environment Architecture .................... 1027

Chapter 1. Storage Model ............................................. 1029
1.1 Definitions .......................................................... 1029
1.2 Introduction ........................................................ 1030
1.3 Virtual Storage ....................................................... 1031
1.4 Single-Copy Atomicity ............................................. 1031
1.5 Cache Model ........................................................ 1032
1.6 Storage Control Attributes ..................................... 1032
1.6.1 Write Through Required .......................................... 1033
1.6.2 Caching Inhibited ................................................ 1033
1.6.3 Memory Coherence Required .................................... 1033
1.6.4 Guarded ........................................................... 1034
1.7 Shared Storage ....................................................... 1034
1.7.1 Storage Access Ordering ......................................... 1034
1.7.1.1 Storage Ordering of Copy/Paste-Initiated Data Transfers ........ 1034
1.7.1.2 Storage Ordering of Stores to Persistent Storage ........... 1037
1.7.1.3 Storage Ordering of I/O Accesses ............................. 1038
1.7.2 Atomic Update ..................................................... 1038
1.7.2.1 Reservations .................................................. 1039
1.7.2.2 Forward Progress ............................................ 1040
1.8 Instruction Storage .................................................. 1041
Chapter 2. Instruction Restart. 1045

Chapter 3. Management of Shared Resources 1047
  3.1 Program Priority Registers 1047
  3.2 "or" Instruction 1048

Chapter 4. Storage Control Instructions 1049
  4.1 Parameters Useful to Application Programs 1049
  4.2 Data Stream Control Register (DSCR) 1049
  4.3 Cache Management Instructions 1051
  4.3.1 Instruction Cache Instructions 1052
  4.3.2 Data Cache Instructions 1053
  4.3.2.1 Obsolete Data Cache Instructions 1065
  4.3.3 "or" Instruction 1066
  4.4 Copy-Paste Facility 1067
  4.5 Atomic Memory Operations 1070
  4.5.1 Load Atomic 1070
  4.5.2 Store Atomic 1074
  4.6 Synchronization Instructions 1076
  4.6.1 Instruction Synchronize Instruction 1076
  4.6.2 Load And Reserve and Store Conditional Instructions 1076
  4.6.2.1 64-Bit Load And Reserve and Store Conditional Instructions 1082
  4.6.2.2 128-bit Load And Reserve and Store Conditional Instructions 1084
  4.6.3 Memory Barrier Instructions 1086
  4.6.4 Wait Instruction 1090

Chapter 5. Time Base 1093
  5.1 Time Base Instructions 1094

Chapter 6. Event-Based Branch Facility 1097
  6.1 Event-Based Branch Overview 1097
  6.2 Event-Based Branch Registers 1097
  6.2.1 Branch Event Status and Control Register 1097
  6.2.2 Event-Based Branch Handler Register 1099
  6.2.3 Event-Based Branch Return Register 1099
  6.3 Event-Based Branch Instructions 1100

Chapter 7. Branch History Rolling Buffer 1101
  7.1 Branch History Rolling Buffer Entry Format 1102
  7.2 Branch History Rolling Buffer Instructions 1103

Appendix A. Assembler Extended Mnemonics 1105
  A.1 Data Cache Block Touch [for Store] Mnemonics 1105
  A.2 Data Cache Block Flush Mnemonics 1105
  A.3 Or Mnemonics 1105
  A.4 Load And Reserve Mnemonics 1105
  A.5 Synchronize Mnemonics 1106
  A.6 Wait Mnemonics 1106
  A.7 Move To/From Time Base Mnemonics 1106
  A.8 Return From Event-Based Branch Mnemonic 1106

Appendix B. Programming Examples for Sharing Storage 1107
  B.1 Atomic Update Primitives 1107
  B.2 Lock Acquisition and Release, and Related Techniques 1109
  B.2.1 Lock Acquisition and Import Barriers 1109
  B.2.1.1 Acquire Lock and Import Shared Storage 1109
  B.2.1.2 Obtain Pointer and Import Shared Storage 1109
  B.2.2 Lock Release and Export Barriers 1110
  B.2.2.1 Export Shared Storage and Release Lock 1110
  B.2.2.2 Export Shared Storage and Release Lock using lwsync 1110
  B.2.3 Safe Fetch 1110
  B.3 List Insertion 1111
  B.4 Notes 1111

Book III:

Power ISA Operating Environment Architecture 1113

Chapter 1. Introduction 1115
  1.1 Overview 1115
  1.2 Document Conventions 1115
  1.2.1 Definitions and Notation 1115
1.2.2 Reserved Fields ................. 1116
1.2.3 Deviations from the Sequential Execution Model ................. 1117
1.2.4 Restricting Out-of-Order Execution ................................ 1117
1.3 General Systems Overview .................. 1117
1.4 Exceptions .......................... 1117
1.5 Synchronization ...................... 1118
1.5.1 Context Synchronization .......... 1118
1.5.2 Execution Synchronization ...... 1119

Chapter 2. Logical Partitioning (LPAR) and Thread Control .... 1121
2.1 Overview ............................ 1121
2.2 Logical Partitioning Control Register (LPCCR) ....................... 1121
2.3 Hypervisor Real Mode Offset Register (HRMOR) ...................... 1126
2.4 Logical Partition Identification Register (LPIDR) .................... 1126
2.5 Processor Compatibility Register (PCR) .......................... 1126
2.6 Other Hypervisor Resources ........ 1136
2.7 Sharing Hypervisor and Ultravisor Resources .............. 1137
2.8 Sub-Processors ...................... 1137
2.9 Thread Identification Register (TIR) .......................... 1137
2.10 Hypervisor Interrupt Little-Endian (HILE) Bit .................. 1138

Chapter 3. Ultravisor and Secure Memory Facility (SMF) .... 1139
3.1 Overview ............................ 1139
3.2 Ultravisor Real Mode Offset Register (URMOR) ....................... 1140
3.3 Ultravisor Interrupt Little-Endian (UILE) Bit ...................... 1140
3.4 Secure Memory Facility Control Register (SMFCTRL) ................ 1140
3.4.1 Enabling SMF and Secure Memory Enforcement ................ 1141

Chapter 4. Branch Facility ......... 1143
4.1 Branch Facility Overview .................. 1143
4.2 Branch Facility Registers .............. 1143
4.2.1 Machine State Register .............. 1143
4.2.2 Processor Stop Status and Control Register (PSSCR) ........... 1147
4.3 Branch Facility Instructions ........... 1150
4.3.1 System Linkage Instructions ........ 1150
4.3.2 Power-Saving Mode ................ 1154
4.3.2.1 Power-Saving Mode Instruction .... 1155
4.3.2.2 Entering and Exiting Power-Saving Mode ................. 1155
4.4 Event-Based Branch Facility and Instruction .................... 1157

Chapter 5. Fixed-Point Facility .... 1159
5.1 Fixed-Point Facility Overview ........... 1159
5.2 Special Purpose Registers .............. 1159
5.3 Fixed-Point Facility Registers .......... 1159
5.3.1 Processor Version Register .......... 1159
5.3.2 Processor Identification Register .......... 1159
5.3.3 Process Identification Register ........ 1160
5.3.4 Control Register .................... 1160
5.3.5 Program Priority Register .......... 1160
5.3.6 Problem State Priority Boost Register .......... 1160
5.3.7 Relative Priority Register .......... 1161
5.3.8 Software-use SPRs .............. 1161
5.4 Fixed-Point Facility Instructions .... 1163
5.4.1 Fixed-Point Load and Store Caching Inhibited Instructions .......... 1163
5.4.2 OR Instruction .................... 1166
5.4.3 OR Immediate Instruction ............ 1166
5.4.4 Move To/From System Register Instructions .......... 1166

Chapter 6. Storage Control ......... 1177
6.1 Overview ............................ 1177
6.2 Storage Exceptions .............. 1177
6.3 Instruction Fetch ..................... 1177
6.3.1 Implicit Branch .............. 1177
6.3.2 Address Wrapping Combined with Changing MSR Bit SF .......... 1177
6.4 Data Access ..................... 1178
6.5 Performing Operations Out-of-Order .......... 1178
6.6 Invalid Real Address .......... 1178
6.7 Storage Addressing .......... 1179
6.7.1 32-Bit Mode .......... 1179
6.7.2 Virtualized Partition Memory (VPM) Mode .......... 1180
6.7.3 Ultravisor Real, Hypervisor Real, and Virtual Real Addressing Modes ........ 1180
6.7.3.1 Ultravisor/Hypervisor Offset Real Mode Address .......... 1180
6.7.3.2 Storage Control Attributes for Accesses in Ultravisor and Hypervisor Real Addressing Modes .......... 1181
6.7.3.2.1 Hypervisor Real Mode Storage Control .......... 1181
6.7.3.3 Virtual Real Mode Addressing Mechanism .......... 1182
6.7.3.4 Storage Control Attributes for Implicit Storage Accesses .......... 1183
6.7.4 Definitions ........................ 1183
6.7.5 Address Ranges Having Defined Uses ........................................ 1184

6.7.5.1 Effective Address Space Structure for Radix-using Partitions .... 1184

6.7.6 In-Memory Tables ..................................................... 1185

6.7.6.1 Partition Table .................................................... 1185

6.7.6.2 Process Table ..................................................... 1187

6.7.7 Address Translation Overview .................................... 1188

6.7.8 Segment Translation .................................................. 1190

6.7.8.1 Segment Lookaside Buffer (SLB) ................................. 1190

6.7.8.2 SLB Search ....................................................... 1192

6.7.8.3 Segment Table Description and Search ........................... 1192

6.7.8.3.1 Primary Hash for 256MB Segment .......................... 1192

6.7.8.3.2 Primary Hash for 1TB Segment ............................... 1192

6.7.8.3.3 Secondary Hash for 256MB Segment ....................... 1192

6.7.8.3.4 Secondary Hash for 1TB Segment ............................. 1193

6.7.9 Hashed Page Table Translation .................................. 1193

6.7.9.1 Hashed Page Table .................................................. 1195

6.7.9.2 Page Table Search ................................................ 1196

6.7.10 Radix Tree Translation .............................................. 1198

6.7.10.1 Radix Tree Page Directory Entry ............................... 1199

6.7.10.2 Radix Tree Page Table Entry .................................. 1200

6.7.10.3 Nested Translation .............................................. 1200

6.7.11 Translation Process .................................................. 1202

6.7.11.1 Fully-Qualified Address ........................................ 1202

6.7.11.2 Finding the Page Tables ...................................... 1202

6.7.11.3 Obtaining Host Real Address, Radix on Radix ............... 1203

6.7.11.4 Obtaining Host Real Address, HPT ............................... 1204

6.7.12 Reference and Change Recording .................................. 1204

6.7.13 Storage Protection ...................................................... 1208

6.7.13.1 Virtual Page Class Key Protection ............................. 1208

6.7.13.2 Basic Storage Protection, Address Translation Enabled .... 1212

6.7.13.3 Basic Storage Protection, Address Translation Disabled ..... 1213

6.7.13.4 Radix Tree Translation Storage Protection ....................... 1213

6.7.13.5 Secure Memory Protection ...................................... 1214

6.8 Storage Control Attributes ............................................. 1214

6.8.1 Guarded Storage ..................................................... 1215

6.8.1.1 Out-of-Order Accesses to Guarded Storage ....................... 1215

6.8.2 Storage Control Bits .................................................. 1215

6.8.2.1 Storage Control Bit Restrictions ................................. 1216

6.8.2.2 Altering the Storage Control Bits ............................... 1216

6.9 Storage Control Instructions .......................................... 1218

6.9.1 Cache Management Instructions ................................... 1218

6.9.2 Synchronize Instruction ............................................. 1218

6.9.3 Lookaside Buffer Management ..................................... 1219

6.9.3.1 Thread-Specific Segment Translations ......................... 1220

6.9.3.2 SLB Management Instructions ................................. 1220

6.9.3.3 TLB Management Instructions .................................. 1231

6.10 Translation Table Update Synchronization Requirements ............. 1241

6.10.1 Translation Table Updates ....................................... 1242

6.10.1.1 Adding a Page Table Entry ................................. 1243

6.10.1.2 Modifying a Translation Table Entry .......................... 1243

Chapter 7. Interrupts ...................................................... 1247

7.1 Overview ................................................................. 1247

7.2 Interrupt Registers .................................................... 1247

7.2.1 Machine Status Save/Restore Registers ........................... 1247

7.2.2 Hypervisor Machine Status Save/Restore Registers ............... 1247

7.2.3 Ultravisor Machine Status Save/Restore Registers ............... 1247

7.2.4 Access Segment Descriptor Register ............................. 1248

7.2.5 Data Address Register ............................................. 1248

7.2.6 Hypervisor Data Address Register ............................... 1248

7.2.7 Data Storage Interrupt Status Register .......................... 1248

7.2.8 Hypervisor Data Storage Interrupt Status Register ............ 1249

7.2.9 Hypervisor Emulation Instruction Register ....................... 1249

7.2.10 Hypervisor Maintenance Exception Register .................... 1249

7.2.11 Hypervisor Maintenance Exception Enable Register ............ 1249

7.2.12 Facility Status and Control Register ............................ 1250

7.2.13 Hypervisor Facility Status and Control Register ............... 1251

7.3 Interrupt Synchronization .............................................. 1255

7.4 Interrupt Classes ....................................................... 1255

7.4.1 Precise Interrupt .................................................... 1255

7.4.2 Imprecise Interrupt .................................................. 1255

7.4.3 Interrupt Processing ................................................ 1256

7.4.4 Implicit alteration of HSRR0 and HSRR1 .......................... 1258
## 7.5 Interrupt Definitions

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.5.1</td>
<td>System Reset Interrupt</td>
<td>1262</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Machine Check Interrupt</td>
<td>1264</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Data Storage Interrupt (DSI)</td>
<td>1266</td>
</tr>
<tr>
<td>7.5.4</td>
<td>Data Segment Interrupt</td>
<td>1268</td>
</tr>
<tr>
<td>7.5.5</td>
<td>Instruction Storage Interrupt (ISI)</td>
<td>1269</td>
</tr>
<tr>
<td>7.5.6</td>
<td>Instruction Segment Interrupt</td>
<td>1270</td>
</tr>
<tr>
<td>7.5.7</td>
<td>External Interrupt</td>
<td>1270</td>
</tr>
<tr>
<td>7.5.7.1</td>
<td>Direct External Interrupt</td>
<td>1270</td>
</tr>
<tr>
<td>7.5.8</td>
<td>Alignment Interrupt</td>
<td>1271</td>
</tr>
<tr>
<td>7.5.9</td>
<td>Program Interrupt</td>
<td>1272</td>
</tr>
<tr>
<td>7.5.10</td>
<td>Floating-Point Unavailable Interrupt</td>
<td>1274</td>
</tr>
<tr>
<td>7.5.11</td>
<td>Decrementer Interrupt</td>
<td>1275</td>
</tr>
<tr>
<td>7.5.12</td>
<td>Hypervisor Decrementer Interrupt</td>
<td>1275</td>
</tr>
<tr>
<td>7.5.13</td>
<td>Directed Privileged Doorbell Interrupt</td>
<td>1275</td>
</tr>
<tr>
<td>7.5.14</td>
<td>System Call Interrupt</td>
<td>1275</td>
</tr>
<tr>
<td>7.5.15</td>
<td>Trace Interrupt</td>
<td>1276</td>
</tr>
<tr>
<td>7.5.16</td>
<td>Hypervisor Data Storage Interrupt (HDSI)</td>
<td>1277</td>
</tr>
<tr>
<td>7.5.17</td>
<td>Hypervisor Instruction Storage Interrupt (HISI)</td>
<td>1280</td>
</tr>
<tr>
<td>7.5.18</td>
<td>Hypervisor Emulation Assistance Interrupt</td>
<td>1282</td>
</tr>
<tr>
<td>7.5.19</td>
<td>Hypervisor Maintenance Interrupt</td>
<td>1284</td>
</tr>
<tr>
<td>7.5.20</td>
<td>Directed Hypervisor Doorbell Interrupt</td>
<td>1284</td>
</tr>
<tr>
<td>7.5.21</td>
<td>Hypervisor Virtualization Interrupt</td>
<td>1285</td>
</tr>
<tr>
<td>7.5.22</td>
<td>Performance Monitor Interrupt</td>
<td>1285</td>
</tr>
<tr>
<td>7.5.23</td>
<td>Vector Unavailable Interrupt</td>
<td>1285</td>
</tr>
<tr>
<td>7.5.24</td>
<td>VSX Unavailable Interrupt</td>
<td>1285</td>
</tr>
<tr>
<td>7.5.25</td>
<td>Facility Unavailable Interrupt</td>
<td>1286</td>
</tr>
<tr>
<td>7.5.26</td>
<td>Hypervisor Facility Unavailable Interrupt</td>
<td>1286</td>
</tr>
<tr>
<td>7.5.27</td>
<td>System Call Vectored Interrupt</td>
<td>1287</td>
</tr>
<tr>
<td>7.5.28</td>
<td>Directed Ultravisor Doorbell Interrupt</td>
<td>1288</td>
</tr>
<tr>
<td>7.5.29</td>
<td>Event-Based Branch Exception Order</td>
<td>1288</td>
</tr>
<tr>
<td>7.5.30</td>
<td>Interrupt Priorities</td>
<td>1290</td>
</tr>
</tbody>
</table>

## Chapter 8. Timer Facilities

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Overview</td>
<td>1295</td>
</tr>
<tr>
<td>8.2</td>
<td>Time Base (TB)</td>
<td>1295</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Writing the Time Base</td>
<td>1296</td>
</tr>
<tr>
<td>8.3</td>
<td>Virtual Time Base</td>
<td>1296</td>
</tr>
<tr>
<td>8.4</td>
<td>Decrementer</td>
<td>1297</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Writing and Reading the Decrementer</td>
<td>1298</td>
</tr>
<tr>
<td>8.5</td>
<td>Hypervisor Decrementer</td>
<td>1298</td>
</tr>
<tr>
<td>8.6</td>
<td>Processor Utilization of Resources Register (PURR)</td>
<td>1298</td>
</tr>
<tr>
<td>8.7</td>
<td>Scaled Processor Utilization of Resources Register (SPURR)</td>
<td>1299</td>
</tr>
<tr>
<td>8.8</td>
<td>Instruction Counter</td>
<td>1300</td>
</tr>
</tbody>
</table>

## Chapter 9. Debug Facilities

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Overview</td>
<td>1301</td>
</tr>
<tr>
<td>9.2</td>
<td>Come-From Address Register</td>
<td>1301</td>
</tr>
<tr>
<td>9.3</td>
<td>Completed Instruction Address Breakpoint</td>
<td>1301</td>
</tr>
<tr>
<td>9.4</td>
<td>Data Address Watchpoint</td>
<td>1302</td>
</tr>
</tbody>
</table>

## Chapter 10. Performance Monitor Facility

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>Overview</td>
<td>1305</td>
</tr>
<tr>
<td>10.2</td>
<td>Performance Monitor Operation</td>
<td>1305</td>
</tr>
<tr>
<td>10.3</td>
<td>No-op Instructions Reserved for the Performance Monitor</td>
<td>1306</td>
</tr>
<tr>
<td>10.4</td>
<td>Performance Monitor Facility Registers</td>
<td>1306</td>
</tr>
<tr>
<td>10.4.1</td>
<td>Performance Monitor SPR Numbers</td>
<td>1307</td>
</tr>
<tr>
<td>10.4.2</td>
<td>Performance Monitor Counters</td>
<td>1307</td>
</tr>
<tr>
<td>10.4.2.1</td>
<td>Event Counting and Sampling</td>
<td>1307</td>
</tr>
<tr>
<td>10.4.3</td>
<td>Threshold Event Counter</td>
<td>1308</td>
</tr>
<tr>
<td>10.4.4</td>
<td>Monitor Mode Control Register 0</td>
<td>1309</td>
</tr>
<tr>
<td>10.4.5</td>
<td>Monitor Mode Control Register 1</td>
<td>1313</td>
</tr>
<tr>
<td>10.4.6</td>
<td>Monitor Mode Control Register 2</td>
<td>1316</td>
</tr>
<tr>
<td>10.4.7</td>
<td>Monitor Mode Control Register A</td>
<td>1317</td>
</tr>
<tr>
<td>10.4.8</td>
<td>Sampled Instruction Address Register</td>
<td>1320</td>
</tr>
<tr>
<td>10.4.9</td>
<td>Sampled Data Address Register</td>
<td>1320</td>
</tr>
<tr>
<td>10.4.10</td>
<td>Sampled Instruction Event Register</td>
<td>1320</td>
</tr>
</tbody>
</table>
Book I: Power ISA User Instruction Set Architecture
Chapter 1. Introduction

1.1 Overview
This chapter describes computation modes, document conventions, a processor overview, instruction formats, storage addressing, and instruction fetching.

1.2 Instruction Mnemonics and Operands
The description of each instruction includes the mnemonic and a formatted list of operands. Some examples are the following.

```
stw RS,D(RA)
addis RT,RA,SI
```
Power ISA-compliant Assemblers will support the mnemonics and operand lists exactly as shown. They should also provide certain extended mnemonics, such as the ones described either in Appendix C of Book I or in the instruction description, if extended mnemonics are provided for the instruction. Assemblers will support extended mnemonics having a reduced number of operands using the specified default values for any operands omitted from the base form.

1.3 Document Conventions

1.3.1 Definitions
The following definitions are used throughout this document.

- **program**
  A sequence of related instructions.

- **application program**
  A program that uses only the instructions and resources described in Books I and II.

- **processor**
  The hardware component that implements the instruction set, storage model, and other facilities defined in the Power ISA architecture, and executes the instructions specified in a program.

- **octword, quadword, doubleword, word, halfword, byte, and nibble**
  256 bits, 128 bits, 64 bits, 32 bits, 16 bits, 8 bits, and 4 bits, respectively.

- **positive**
  Means greater than zero.

- **negative**
  Means less than zero.

- **floating-point single format (or simply single format)**
  Refers to the representation of a single-precision binary floating-point value in a register or storage.

- **floating-point double format (or simply double format)**
  Refers to the representation of a double-precision binary floating-point value in a register or storage.

- **system library program**
  A component of the system software that can be called by an application program using a Branch instruction.

- **system service program**
  A component of the system software that can be called by an application program using a System Call or System Call Vectored instruction.

- **system trap handler**
  A component of the system software that receives control when the conditions specified in a Trap instruction are satisfied.

- **system error handler**
  A component of the system software that receives control when an error occurs. The system error handler includes a component for each of the various kinds of error. These error-specific components are referred to as the system alignment error handler, the system data storage error handler, etc.

- **latency**
  Refers to the interval from the time an instruction begins execution until it produces a result that is available for use by a subsequent instruction.
■ unavailable
  Refers to a resource that cannot be used by the program. For example, storage is unavailable if access to it is denied. See Book III.

■ undefined value
  May vary between implementations, and between different executions on the same implementation, and similarly for register contents, storage contents, etc., that are specified as being undefined.

■ boundedly undefined
  The results of executing a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary finite sequence of instructions (none of which yields boundedly undefined results) in the state the processor was in before executing the given instruction. Boundedly undefined results may include the presentation of inconsistent state to the system error handler as described in Section 1.8.1 of Book II. Boundedly undefined results for a given instruction may vary between implementations, and between different executions on the same implementation.

■ “must”
  If software violates a rule that is stated using the word “must” (e.g., “this field must be set to 0”), the results are boundedly undefined unless otherwise stated.

■ sequential execution model
  The model of program execution described in Section 2.2, “Instruction Execution Order” on page 33.

1.3.2 Notation

The following notation is used throughout the Power ISA documents.

■ All numbers are decimal unless specified in some special way.
  - 0bnnnn means a number expressed in binary format.
  - 0xnnnn means a number expressed in hexadecimal format.

Underscores may be used between digits.

■ RT, RA, R1, ... refer to General Purpose Registers.

■ FRT, FRA, FR1, ... refer to Floating-Point Registers.

■ FRTp, FRAp, FRBp, ... refer to an even-odd pair of Floating-Point Registers. Values must be even, otherwise the instruction form is invalid.

■ VRT, VRA, VR1, ... refer to Vector Registers.

■ (x) means the contents of register x, where x is the name of an instruction field. For example, (RA) means the contents of register RA, and (FRA) means the contents of register FRA, where RA and FRA are instruction fields. Names such as LR and CTR denote registers, not fields, so parentheses are not used with them. Parentheses are also omitted when register x is the register into which the result of an operation is placed.

■ (RA(0)) means the contents of register RA if the RA field has the value 1-31, or the value 0 if the RA field is 0.

■ Bytes in registers, instructions, fields, and bit strings are numbered from left to right, starting with byte 0 (most significant).

■ Bits in registers, instructions, fields, and bit strings are specified as follows. In the last three items (definition of $X_p$ etc.), if $X$ is a field that specifies a GPR, FPR, or VR (e.g., the RS field of an instruction), the definitions apply to the register, not to the field.
  - Bits in instructions, fields, and bit strings are numbered from left to right, starting with bit 0
  - For all registers except the Vector registers, bits in registers that are less than 64 bits start with bit number 64-L, where L is the register length; for the Vector registers, bits in registers that are less than 128 bits start with bit number 128-L.
  - The leftmost bit of a sequence of bits is the most significant bit of the sequence.
  - $X_p$ means bit $p$ of register/instruction/field/bit_string $X$.
  - $X_{p:q}$ means bits $p$ through $q$ of register/instruction/field/bit_string $X$.
  - $X_{p\ q\ ...}$ means bits $p$, $q$, ... of register/instruction/field/bit_string $X$.

■ ¬(RA) means the one’s complement of the contents of register RA.

■ A period (.) as the last character of an instruction mnemonic means that the instruction records status information in certain fields of the Condition Register as a side effect of execution.

■ The symbol || is used to describe the concatenation of two values. For example, 010 || 111 is the same as 010111.

■ $x^n$ means $x$ raised to the $n$th power.

■ $x^n$ means the replication of $x$, $n$ times (i.e., $x$ concatenated to itself $n$-1 times). $n^0$ and $n^1$ are special cases:
  - $n^0$ means a field of $n$ bits with each bit equal to 0. Thus $5^0$ is equivalent to 0b00000.
  - $n^1$ means a field of $n$ bits with each bit equal to 1. Thus $5^1$ is equivalent to 0b11111.

■ Each bit and field in instructions, and in status and control registers (e.g., XER, FPSCR) and Special
1.3.3 Reserved Fields, Reserved Values, and Reserved SPRs

Reserved fields in instructions are ignored by the processor.

In some cases a defined field of an instruction has certain values that are reserved. This includes cases in which the field is shown in the instruction layout as containing a particular value; in such cases all other values of the field are reserved. In general, if an instruction is coded such that a defined field contains a reserved value the instruction form is invalid; see Section 1.8.2 on page 24. The only exception to the preceding rule is that it does not apply to Reserved and Illegal classes of instructions (see Section 1.6.3) or to portions of defined fields that are specified, in the instruction description, as being treated as reserved fields.

To maximize compatibility with future architecture extensions, software must ensure that reserved fields in instructions contain zero and that defined fields of instructions do not contain reserved values.

The handling of reserved bits in System Registers (e.g., XER, FPSCR) depends on whether the processor is in problem state. Unless otherwise stated, software is permitted to write any value to such a bit. In problem state, a subsequent reading of the bit returns 0 regardless of the value written; in privileged states, a subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

In some cases, a defined field of a System Register has certain values that are reserved. Software must not set a defined field of a System Register to a reserved value. References elsewhere in this document to a defined field (in an instruction or System Register) that has reserved values assume the field does not contain a reserved value, unless otherwise stated or obvious from context.

In some cases, a given bit of a System Register is specified to be set to a constant value by a given instruction or event. Unless otherwise stated or obvious from context, software should not depend on this constant value because the bit may be assigned a meaning in a future version of the architecture.

The reserved SPRs include SPRs 808, 809, 810, and 811. \texttt{mtspr} and \texttt{mfspr} instructions specifying these SPRs are treated as no-ops. Reserved SPRs are provided in the architecture to anticipate the eventual adoption of performance hint functionality that must be controlled by SPRs. Control of these capabilities using reserved SPRs will allow software to use these new capabilities on new implementations that support them while remaining compatible with existing implementations that may not support the new functionality.
Reserved SPRs are not assigned names. There are no individual descriptions of reserved SPRs in this document.

**Assembler Note**

Assemblers should report uses of reserved values of defined fields of instructions as errors.

**Programming Note**

It is the responsibility of software to preserve bits that are now reserved in System Registers, because they may be assigned a meaning in some future version of the architecture.

In order to accomplish this preservation in implementation-independent fashion, software should do the following.

- Initialize each such register supplying zeros for all reserved bits.
- Alter (defined) bit(s) in the register by reading the register, altering only the desired bit(s), and then writing the new value back to the register.

The XER and FPSCR are partial exceptions to this recommendation. Software can alter the status bits in these registers, preserving the reserved bits, by executing instructions that have the side effect of altering the status bits. Similarly, software can alter any defined bit in the FPSCR by executing a Floating-Point Status and Control Register instruction. Using such instructions is likely to yield better performance than using the method described in the second item above.

### 1.3.4 Description of Instruction Operation

Instruction descriptions (including related material such as the introduction to the section describing the instructions) mention that the instruction may cause a system error handler to be invoked, under certain conditions, if and only if the system error handler may treat the case as a programming error. (An instruction may cause a system error handler to be invoked under other conditions as well; see Chapter 7 of Book III).

A formal description is given of the operation of each instruction. In addition, the operation of most instructions is described by a semiformal language at the register transfer level (RTL). This RTL uses the notation given below, in addition to the notation described in Section 1.3.2. Some of this notation is also used in the formal descriptions of instructions. RTL notation not summarized here should be self-explanatory.

The RTL descriptions cover the normal execution of the instruction, except that “standard” setting of status registers, such as the Condition Register, is not shown. (“Non-standard” setting of these registers, such as the setting of the Condition Register by the Compare instructions, is shown.) The RTL descriptions do not cover cases in which the system error handler is invoked, or for which the results are boundedly undefined.

The RTL descriptions specify the architectural transformation performed by the execution of an instruction. They do not imply any particular implementation.

#### Notation Meaning

- Assignment
- Assignment of an instruction effective address. In 32-bit mode the high-order 32 bits of the 64-bit target address are set to 0.
- NOT logical operator
- Two’s complement addition
- Two’s complement subtraction, unary minus
- Multiplication
- Signed-integer multiplication
- Unsigned-integer multiplication
- Division
- Division, with result truncated to integer
- Remainder of integer division
- Square root
- Equals, Not Equals relations
- Signed comparison relations
- Unsigned comparison relations
- Unordered comparison relation
- AND, OR logical operators
- Exclusive OR, Equivalence logical operators

#### ABS(x)
Absolute value of x

#### BCD_TO_DPD(x)
The low-order 24 bits of x contain six, 4-bit BCD fields which are converted to two declets; each set of two declets is placed into the low-order 20 bits of the result. See Section B.1, “BCD-to-DPD Translation”.

#### CEIL(x)
Least integer \( \geq x \)

#### DOUBLE(x)
Result of converting x from floating-point single format to floating-point double format, using the model shown on page 149

#### DPD_TO_BCD(x)
The low-order 20 bits of x contain two declets which are converted to six, 4-bit BCD fields; each set of six, 4-bit BCD fields is placed into the low-order 24 bits of the result. See Section B.2, “DPD-to-BCD Translation”.

#### EXTS(x)
Result of extending x on the left with sign bits

#### FLOOR(x)
Greatest integer \( \leq x \)

#### GPR(x)
General Purpose Register x

#### MASK(x, y)
Mask having 1s in positions x through y (wrapping if \( x > y \)) and 0s elsewhere

Assemblers should report uses of reserved values of defined fields of instructions as errors.
MEM(x, y) Contents of a sequence of y bytes of storage. The sequence depends on the byte ordering used for storage access, as follows.

Big-Endian byte ordering:
The sequence starts with the byte at address x and ends with the byte at address x+y-1.

Little-Endian byte ordering:
The sequence starts with the byte at address x+y-1 and ends with the byte at address x.

MEM\text{metadata}(x,y) Metadata associated with MEM(x,y).

ROTL_{64}(x, y) Result of rotating the 64-bit value x left y positions

ROTL_{32}(x, y) Result of rotating the 64-bit value x|x left y positions, where x is 32 bits long

SINGLE(x) Result of converting x from floating-point double format to floating-point single format, using the model shown on page 154

SPR(x) Special Purpose Register x

TRAP Invoke the system trap handler

Characterization
Reference to the setting of status bits, in a standard way that is explained in the text

undefined An undefined value.

CIA Current Instruction Address, which is the 64-bit address of the instruction being described by a sequence of RTL. Used by relative branches to set the Next Instruction Address (NIA), and by Branch instructions with LK=1 to set the Link Register.

NIA Next Instruction Address, which is the 64-bit address of the next instruction to be executed. For a successful branch, the next instruction address is the branch target address: in RTL, this is indicated by assigning a value to NIA. For other instructions that cause non-sequential instruction fetching (see Book III), the RTL is similar.

For instructions that do not branch, and do not otherwise cause instruction fetching to be non-sequential, the next instruction address is CIA+4. Does not correspond to any architected register.

if... then... else... Conditional execution, indenting shows range; else is optional.

do Do loop, indenting shows range. “To” and/or “by” clauses specify incrementing an iteration variable, and a “while” clause gives termination conditions.

leave Leave innermost do loop, or do loop described in leave statement.
The precedence rules for RTL operators are summarized in Table 1. Operators higher in the table are applied before those lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. (For example, $-\cdot$ associates from left to right, so $a-b-c = (a-b)-c$.) Parentheses are used to override the evaluation order implied by the table or to increase clarity; parenthesized expressions are evaluated before serving as operands.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscript, function evaluation</td>
<td>left to right</td>
</tr>
<tr>
<td>pre-superscript (replication),</td>
<td>right to left</td>
</tr>
<tr>
<td>post-superscript (exponentiation)</td>
<td>right to left</td>
</tr>
<tr>
<td>unary $\neg$, $\bar{\cdot}$</td>
<td>right to left</td>
</tr>
<tr>
<td>$\times$, $\div$</td>
<td>left to right</td>
</tr>
<tr>
<td>$+$, $-$,</td>
<td>left to right</td>
</tr>
<tr>
<td>$</td>
<td></td>
</tr>
<tr>
<td>$=, \neq, &lt;, \leq, \geq, \ltu, \gtu, ?$</td>
<td>left to right</td>
</tr>
<tr>
<td>$&amp;$, $\oplus$, $\equiv$</td>
<td>left to right</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$:$ (range)</td>
<td>none</td>
</tr>
<tr>
<td>$\leftarrow$, $\rightarrow$</td>
<td>none</td>
</tr>
</tbody>
</table>

### 1.3.5 Phased-Out Facilities

#### Phased-Out Facilities

These are facilities and instructions that, in some future version of the architecture, will be dropped out of the architecture. System developers should develop a migration plan to eliminate use of them in new systems. These facilities are marked with a [Phased-Out] marker.

Phased-Out facilities and instructions must be implemented.

---

**Warning:** Instructions and facilities being phased out of the architecture are likely to perform poorly on future implementations. New programs should not use them.
1.4 Processor Overview

The basic classes of instructions are as follows:

- branch instructions (Chapter 2)
- GPR-based scalar fixed-point instructions (Chapter 3)
- FPR-based scalar floating-point instructions (Chapter 4)
- FPR-based scalar decimal floating-point instructions (Chapter 5)
- VR-based vector fixed-point and floating-point instructions (Chapter 6)
- VSR-based scalar and vector floating-point instructions (Chapter 7)

Scalar fixed-point instructions operate on byte, halfword, word, doubleword, and quadword operands, where each operand is contained in a GPR (or a pair of GPRs for quadword operands). Vector fixed-point instructions operate on vectors of nibble, byte, halfword, word, doubleword, and quadword operands, where each vector is contained in a VR. Scalar binary floating-point instructions operate on single-precision, double-precision, and quad-precision floating-point operands, where each operand is contained in an FPR or VSR. Scalar decimal floating-point instructions operate on short, long, and extended decimal floating-point operands, where each operand is contained in an FPR (or a pair of FPRs for quadword operands).

Vector floating-point instructions operate on vectors of single-precision and double-precision floating-point operands, where each vector is contained in a VR or VSR.

The Power ISA uses instructions that are four or eight bytes long and are word-aligned. It provides for byte, halfword, word, doubleword, and quadword operand loads and stores between storage and a set of 32 General Purpose Registers (GPRs). It provides for byte, halfword, word, doubleword, quadword, and octword operand loads and stores between storage and a set of 64 Vector-Scalar Registers (VSRs).

Signed integers are represented in two’s complement form.

There are no computational instructions that modify storage; instructions that reference storage may reformat the data (e.g., load halfword algebraic). To use a storage operand in a computation and then modify the same or another storage location, the contents of the storage operand must be loaded into a register, modified, and then stored back to the target location. Figure 1 is a logical representation of instruction processing. Figure 2 shows the registers that are defined in Book I. (A few additional registers that are available to application programs are defined in other Books, and are not shown in the figure.)
### Figure 2. Registers that are defined in Book I

#### 1.5 Computation modes

Processors provide two execution modes, 64-bit mode and 32-bit mode. In both of these modes, instructions that set a 64-bit register affect all 64 bits. The computational mode controls how the effective address is interpreted, how Condition Register bits and XER bits are set, how the Link Register is set by Branch instructions in which LK=1, and how the Count Register is tested by Branch Conditional instructions. Nearly all instructions are available in both modes (the only exceptions are a few instructions that are defined in Book III). In both modes, effective address computations use all 64 bits of the relevant registers (General Purpose Registers, Floating-Point Registers, etc.).
1.6 Instruction Formats

Instructions are encoded in either four or eight bytes and are word-aligned. When referring specifically to only one of these two types of instructions, the term “word instruction” is used to refer to instructions that are encoded in four bytes, and the term “prefixed instruction” is used to refer to instructions that are encoded in eight bytes using a prefix.

Bits 0:5 always specify the primary opcode (PO, below). Many instructions also have an extended opcode (XO, below). Some instructions also have a third, expanded opcode (EO, below). The remaining bits of the instruction contain one or more fields as shown below for the different instruction formats.

Since all instructions are word-aligned, whenever instruction addresses are presented to the processor (as in Branch instructions) the low-order two bits are ignored. Similarly, whenever the processor develops an instruction address the low-order two bits are zero.

Prefixed instructions consist of a four-byte prefix followed by a four-byte suffix. As such, the address of a prefixed instruction is the address of its prefix. For some prefixed instructions, the four-byte suffix is a defined word instruction, and the prefix modifies or extends the word instruction’s behavior. For other prefixed instructions, while the suffix may or may not correspond to (i.e., have the same 32-bit binary value as) a defined word instruction, the prefix causes the suffix to be decoded using a different opcode space from that used by defined word instructions.

Prefixed instructions do not cross 64-byte instruction address boundaries. When a prefixed instruction crosses a 64-byte boundary, the system alignment error handler is invoked.

Programming Note

Although instructions that set a 64-bit register affect all 64 bits in both 32-bit and 64-bit modes, operating systems often do not preserve the upper 32-bits of all registers across context switches done in 32-bit mode. For this reason, application programs operating in 32-bit mode should not assume that the upper 32 bits of the GPRs are preserved from instruction to instruction unless the operating system is known to preserve these bits.

Programming Note

The instruction address boundary error can only occur with prefixed instructions. Word instructions are word-aligned (four-byte), and thus cannot cross 64-byte boundaries.

The format diagrams given below show horizontally all valid combinations of instruction fields. See Section 1.6.3, “Instruction Prefix Formats” for definitions of instruction fields defined in the prefix.

Split Field Notation

In some cases an instruction field occupies more than one contiguous sequence of bits, or occupies one contiguous sequence of bits that are used in permuted order. Such a field is called a split field. In the format diagrams given below and in the individual instruction layouts, the name of a split field is shown in small letters, once for each of the contiguous sequences. In the RTL description of an instruction having a split field, and in certain other places where individual bits of a split field are identified, the name of the field in small letters represents the concatenation of the sequences from left to right. In all other places, the name of the field is capitalized and represents the concatenation of the sequences in some order, which need not be left to right, as described for each affected instruction.
1.6.1 Word Instruction Formats

1.6.1.1 A-FORM

0  6  11  16  21  26  31
PO    FRT  /// FRB  ///  XO
PO    FRT  FRA  /// FRC  XO
PO    FRT  FRA  FRB  FRC  XO
PO    RT   RA   RB   BC   XO

Figure 3. A instruction format

1.6.1.2 B-FORM

0  6  11  16  3031
PO    BO   BI   BD

Figure 4. B instruction format

1.6.1.3 D-FORM

0  6  11  16  31
PO    BF / L RA SI
PO    BF / L RA UI
PO    FRT RA D
PO    RS RA D
PO    RT RA D
PO    TO RA SI

Figure 5. D instruction format

1.6.1.4 DQ-FORM

0  6  11  16  2829  31
PO    RTp RA DQ PT
PO    S RA DQ SI XO
PO    T RA DQ SI XO

Figure 6. DQ instruction format

1.6.1.5 DS-FORM

0  6  11  16  3031
PO    FRSp RA DS XO
PO    FRTp RA DS XO
PO    RS RA DS XO
PO    RT RA DS XO
PO    VRS RA DS XO
PO    VRT RA DS XO

Figure 7. DS instruction format

1.6.1.6 DX-FORM

0  6  11  16  26  31
PO    RT  d1  d0  XO

Figure 8. DX instruction format

1.6.1.7 I-FORM

0  6  3031
PO    LI

Figure 9. I instruction format

1.6.1.8 M-FORM

0  6  11  16  21  26  31
PO    RS RA RB MB ME RC
PO    RS RA SH MB ME RC

Figure 10. M instruction format

1.6.1.9 MD-FORM

0  6  11  16  21  27  3031
PO    RS RA sh mb XO a
PO    RS RA sh me XO a

Figure 11. MD instruction format

1.6.1.10 MDS-FORM

0  6  11  16  21  27  31
PO    RS RA RB mb XO
PO    RS RA RB me XO

Figure 12. MDS instruction format

1.6.1.11 SC-FORM

0  6  11  16  20  27  3031
PO    /// /// /// LEV /// 0 1
PO    /// /// /// LEV /// 1 /

Figure 13. SC instruction format

1.6.1.12 VA-FORM

0  6  11  16  2122  26  31
PO    RT RA RB RC XO
PO    VRT VRB / SHB XO
PO    VRT VRB VRC XO

Figure 14. VA instruction format

1.6.1.13 VC-FORM

0  6  11  16  2122  31
PO    VRT VRA VRB XO

Figure 15. VC instruction format
1.6.1.14 VX-FORM

Figure 16. VX instruction format

1.6.1.15 X-FORM

Figure 17. X instruction format
| 0 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| PO | RS | RA | RB | XO | || |
| PO | RSp | RA | RB | XO | 1 || |
| PO | RT | /// | /// | XO | / || |
| PO | RT | /// | RB | XO | / || |
| PO | RT | /// | RB | XO | 1 || |
| PO | RT | /// | L | /// | XO | / || |
| PO | RT | RA | FC | XO | / || |
| PO | RT | RA | NB | XO | / || |
| PO | RT | RA | RB | XO | / || |
| PO | RTp | RA | RB | XO | / || |
| PO | S | RA | /// | XO | / || |
| PO | S | RA | RB | XO | / || |
| PO | T | EO | IMM8 | XO | T || |
| PO | T | RA | /// | XO | T || |
| PO | T | RA | RB | XO | T || |
| PO | TH | RA | RB | XO | / || |
| PO | TO | RA | SI | XO | 1 || |
| PO | TO | RA | RB | XO | / || |
| PO | TO | RA | RB | XO | 1 || |
| PO | VRS | RA | RB | XO | / || |
| PO | VRT | EO | VRB | XO | / || |
| PO | VRT | EO | VRB | XO | RO || |
| PO | VRT | RA | RB | XO | / || |
| PO | VRT | VRA | VRB | XO | / || |
| PO | VRT | VRA | VRB | XO | RO || |

Figure 17. X instruction format
1.6.1.16  XFL-FORM

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>7</th>
<th>1516</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>L</td>
<td>FLM</td>
<td>W</td>
<td>FRB</td>
<td>XO</td>
</tr>
</tbody>
</table>

Figure 18. XFL instruction format

1.6.1.17  XFX-FORM

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>1112</th>
<th>1516</th>
<th>2021</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>///</td>
<td>///</td>
<td>1</td>
<td>///</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RS</td>
<td>0</td>
<td>FXM</td>
<td>/</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RS</td>
<td>1</td>
<td>FXM</td>
<td>/</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RS</td>
<td>spr</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>0</td>
<td>///</td>
<td>/</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>1</td>
<td>FXM</td>
<td>/</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>spr</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>tbr</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
</tbody>
</table>

Figure 19. XFX instruction format

1.6.1.18  XL-FORM

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>9</th>
<th>11</th>
<th>14</th>
<th>16</th>
<th>192021</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>///</td>
<td>///</td>
<td>///</td>
<td>XO</td>
<td>/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>///</td>
<td>///</td>
<td>///</td>
<td>XO</td>
<td>/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>BF</td>
<td>///</td>
<td>BFA</td>
<td>///</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>BO</td>
<td>BI</td>
<td>///</td>
<td>BH</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>BT</td>
<td>BA</td>
<td>BB</td>
<td>XO</td>
<td>/</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 20. XL instruction format

1.6.1.19  XO-FORM

| 0 | 6 | 9 | 10111213141516171819202122232425262728293031 |
|---|---|---|-------------------|-------------------|-------------------|-------------------|-------------------|
| PO | RT | RA | /// | XO | / |
| PO | RT | RA | RB | / | XO | / |
| PO | RT | RA | RB | / | XO | / |

Figure 21. XO instruction format

1.6.1.20  XS-FORM

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>3031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>RS</td>
<td>RA</td>
<td>sh</td>
<td>XO</td>
<td>/</td>
</tr>
</tbody>
</table>

Figure 22. XS instruction format

1.6.1.21  XX2-FORM

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>9</th>
<th>1011121314151621</th>
<th>2526293031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>BF</td>
<td>///</td>
<td>///</td>
<td>B</td>
</tr>
<tr>
<td>PO</td>
<td>BF</td>
<td>DCMX</td>
<td>B</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>EO</td>
<td>B</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>///</td>
<td>B</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>///</td>
<td>UIM</td>
<td>B</td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>dx</td>
<td>B</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>EO</td>
<td>B</td>
<td>XO</td>
</tr>
</tbody>
</table>

Figure 23. XX2 instruction format

1.6.1.22  XX3-FORM

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>9</th>
<th>11</th>
<th>16</th>
<th>2122</th>
<th>24</th>
<th>293031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>AT</td>
<td>///</td>
<td>A</td>
<td>B</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>AT</td>
<td>///</td>
<td>Ap</td>
<td>B</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>BF</td>
<td>///</td>
<td>A</td>
<td>B</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>0</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>0</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>RC</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>XO</td>
<td>/</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 24. XX3 instruction format

1.6.1.23  XX4-FORM

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>262728293031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>XO</td>
</tr>
</tbody>
</table>

Figure 25. XX4 instruction format

1.6.1.24  Z22-FORM

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>9</th>
<th>11</th>
<th>1516</th>
<th>22</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>BF</td>
<td>///</td>
<td>FRA</td>
<td>DCM</td>
<td>XO</td>
<td>/</td>
</tr>
<tr>
<td>PO</td>
<td>BF</td>
<td>///</td>
<td>FRAp</td>
<td>DCM</td>
<td>XO</td>
<td>/</td>
</tr>
<tr>
<td>PO</td>
<td>BF</td>
<td>///</td>
<td>FRAp</td>
<td>DCM</td>
<td>XO</td>
<td>/</td>
</tr>
<tr>
<td>PO</td>
<td>FRT</td>
<td>FRA</td>
<td>SH</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>FRTp</td>
<td>FRAp</td>
<td>SH</td>
<td>XO</td>
<td>/</td>
<td></td>
</tr>
</tbody>
</table>

Figure 26. Z22 instruction format
1.6.1.25 Z23-FORM

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>15</th>
<th>16</th>
<th>21</th>
<th>23</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>FRT ///</td>
<td>F</td>
<td>FRB</td>
<td>%C</td>
<td>XO</td>
<td>%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>FRT</td>
<td>FRA</td>
<td>FRB</td>
<td>%C</td>
<td>XO</td>
<td>%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>FRT</td>
<td>TE</td>
<td>FRB</td>
<td>%C</td>
<td>XO</td>
<td>%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>FRTp ///</td>
<td>F</td>
<td>FRBp</td>
<td>%C</td>
<td>XO</td>
<td>%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>FRTp</td>
<td>FRA</td>
<td>FRBp</td>
<td>%C</td>
<td>XO</td>
<td>%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>FRTp</td>
<td>TE</td>
<td>FRBp</td>
<td>%C</td>
<td>XO</td>
<td>%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>VRT ///</td>
<td>F</td>
<td>VRB</td>
<td>%C</td>
<td>XO</td>
<td>%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>VRT ///</td>
<td>F</td>
<td>VRB</td>
<td>%C</td>
<td>XO</td>
<td>%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 27. Z23 instruction format

1.6.2 Word Instruction Fields

AA (30)

Absolute Address.

0 The immediate field represents an address relative to the current instruction address. For I-form branches the effective address of the branch target is the sum of the LI field sign-extended to 64 bits and the address of the branch instruction. For B-form branches the effective address of the branch target is the sum of the BD field sign-extended to 64 bits and the address of the branch instruction.

1 The immediate field represents an absolute address. For I-form branches the effective address of the branch target is the LI field sign-extended to 64 bits. For B-form branches the effective address of the branch target is the BD field sign-extended to 64 bits.

Formats: B, I

AX,A (29,11:15)

Fields that are concatenated to specify a VSR to be used as a source.

Formats: XX3, XX4

BA (11:15)

Field used to specify a bit in the CR to be used as a source.

Formats: XL

BB (16:20)

Field used to specify a bit in the CR to be used as a source.

Formats: XL

BC (21:25)

Field used to specify a bit in the CR to be used as a source.

Formats: A

BD (16:29)

Immediate field used to specify a 14-bit signed two’s complement branch displacement which is concatenated on the right with 0b00 and sign-extended to 64 bits.

Formats: B

BF (6:8)

Field used to specify one of the CR fields or one of the FPSCR fields to be used as a target.

Formats: D, X, XL, XX2, XX3, Z22

BFA (11:13)

Field used to specify one of the CR fields or one of the FPSCR fields to be used as a source.

Formats: X, XL

BH (19:20)

Field used to specify a hint in the Branch Conditional to Link Register and Branch Conditional to Count Register instructions. The encoding is described in Section 2.4, “Branch Instructions”.

Formats: XL

BHRBE (11:20)

Field used to identify the BHRB entry to be used as a source by the Move From Branch History Rolling Buffer instruction.

Formats: X

BI (11:15)

Field used to specify a bit in the CR to be tested by a Branch Conditional instruction.

Formats: B, XL

BO (6:10)

Field used to specify options for the Branch Conditional instructions. The encoding is described in Section 2.4, “Branch Instructions”.

Formats: B, XL, X, XL

BT (6:10)

Field used to specify a bit in the CR or in the FPSCR to be used as a target.

Formats: XL

BX,B (30,16:20)

Fields that are concatenated to specify a VSR to be used as a source.

Formats: XX2, XX3, XX4

CT (7:10)

Field used in X-form instructions to specify a cache target (see Section 4.3.2 of Book II).

Formats: X
CX,C (28,21:25)
Fields that are concatenated to specify a VSR to be used as a source.
Formats: XX4

D (16:31)
Immediate field used to specify a 16-bit signed two's complement integer which is sign-extended to 64 bits.
Formats: D

d0,d1,d2 (16:25,11:15,31)
Immediate fields that are concatenated to specify a 16-bit signed two's complement integer which is sign-extended to 64 bits.
Formats: DX

dc, dm, dx (25,29,11:15)
Immediate fields that are concatenated to specify Data Class Mask.
Formats: XX2

DCM (16:21)
Immediate field used to specify Data Class Mask.
Formats: Z22

DCMX (9:15)
Immediate field used to specify Data Class Mask.
Formats: X, XX2

DGM (16:21)
Immediate field used as the Data Group Mask.
Formats: Z22

DM (22:23)
Immediate field used by xxpermdi instruction as doubleword permute control.
Formats: XX3

DRM (18:20)
Immediate operand field used to specify new decimal floating-point rounding mode.
Formats: X

DQ (16:27)
Immediate field used to specify a 12-bit signed two's complement integer which is concatenated on the right with 0b0000 and sign-extended to 64 bits.
Formats: DQ

DS (16:29)
Immediate field used to specify a 14-bit signed two's complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits.
Formats: DS

EH (31)
Field used to specify a hint in the Load And Reserve instructions. The meaning is described in Section 4.6.2, “Load And Reserve and Store Conditional Instructions”, in Book II.
Formats: X

EO (11:12)
Expanded opcode field
Formats: X

EO (11:15)
Expanded opcode field
Formats: VX, X, XX2

EX (31)
Field used to specify Inexact form of round to quad-precision integer.
Formats: X

FC (16:20)
Field used to specify the function code in Load/Store Atomic instructions.
Formats: X

FLM (7:14)
Field mask used to identify the FPSCR fields that are to be updated by the mtfsf instruction.
Formats: XFL

FRA (11:15)
Field used to specify a FPR to be used as a source.
Formats: A, X, Z22, Z23

FR Ap (11:15)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.
Formats: X, Z22, Z23

FRB (16:20)
Field used to specify an FPR to be used as a source.
Formats: A, X, XFL, Z23

FRBp (16:20)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.
Formats: X, Z23

FRC (21:25)
Field used to specify an FPR to be used as a source.
Formats: A
FRS (6:10)
Field used to specify an FPR to be used as a source.
Formats: D, X

FRSp (6:10)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.
Formats: DS, X

FRT (6:10)
Field used to specify an FPR to be used as a target.
Formats: A, D, X, Z22, Z23

FRTp (6:10)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a target.
Formats: DS, X, Z22, Z23

FXM (12:19)
Field mask used to identify the CR fields that are to be written by the *mtcrf* and *mtocrf* instructions, or read by the *mfocrf* instruction.
Formats: XFX

IB (16:20)
Immediate field used to specify a 5-bit signed integer.
Formats: MDS

IH (8:10)
Field used to specify a hint in the *SLB Invalidate All* instruction. The meaning is described in Section 6.9.3.2, “SLB Management Instructions”, in Book III.
Formats: X

IMM8 (13:20)
Immediate field used to specify an 8-bit integer.
Formats: X

IS (6:10)
Immediate field used to specify a 5-bit signed integer.
Formats: MDS

L (6)
Field used to specify whether the *mtfsf* instruction updates the entire FPSCR.
Formats: XFL

L (10)
Field used to specify whether a fixed-point Compare instruction is to compare 64-bit numbers or 32-bit numbers.
Field used by the *Compare Range Byte* instruction to indicate whether to compare against 1 or 2 ranges of bytes.
Field used by the *Paste* instruction to indicate whether to zero the metadata.
Formats: D, X

L (15)
Field used by the *Move To Machine State Register* instruction (see Book III).
Field used by the *SLB Invalidate All Global* instruction to specify whether the invalidation is for a process or for a partition (see Section 6.9.3.2 of Book III).
Field used by the *SLB Move From Entry VSID* and *SLB Move From Entry ESID* instructions for implementation-specific purposes.
Formats: X

L (14:15)
Field used by the *Deliver A Random Number* instruction (see Section 3.3.9, “Fixed-Point Arithmetic Instructions”) to choose the random number format.
Formats: X

LEV (20:26)
Field used by the *System Call* instructions.
Formats: SC

LI (6:29)
Immediate field used to specify a 24-bit signed two’s complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits.
Formats: I

LK (31)
LINK bit.
0 Do not set the Link Register.
1 Set the Link Register. The address of the instruction following the *Branch* instruction is placed into the Link Register.
Formats: B, I, XL

MB (21:25)
Field used in M-form instructions to specify the first 1-bit of a 64-bit mask, as described in Section 3.3.14, “Fixed-Point Rotate and Shift Instructions” on page 107.
Formats: M
mb (21:26)
Field used in MD-form and MDS-form instructions to specify the first 1-bit of a 64-bit mask, as described in Section 3.3.14, “Fixed-Point Rotate and Shift Instructions” on page 107.
Formats: MD, MDS

me (21:26)
Field used in MD-form and MDS-form instructions to specify the last 1-bit of a 64-bit mask, as described in Section 3.3.14, “Fixed-Point Rotate and Shift Instructions” on page 107.
Formats: MD, MDS

ME (26:30)
Field used in M-form instructions to specify the last 1-bit of a 64-bit mask, as described in Section 3.3.14, “Fixed-Point Rotate and Shift Instructions” on page 107.
Formats: M

NB (16:20)
Field used to specify the number of bytes to move in an immediate Move Assist instruction.
Formats: X

OE (21)
Field used by XO-form instructions to enable setting OV and SO in the XER.
Formats: XO

PL (14:15)
Field used by the wait instruction to specify pause length.
Formats: X

PO (0:5)
Primary opcode.
Formats: all

PRS (14)
Field used to specify whether to invalidate process- or partition-scoped entries for tlbie[].
Formats: X

RC (21)
RECORD bit.
0 Do not alter the Condition Register.
1 Set Condition Register Field 0 or Field 1 as described in Section 2.3.1, “Condition Register” on page 34.
Formats: A, M, MD, MDS, VA, X, XO, XS

Rc (31)
RECORD bit.
0 Do not alter the Condition Register.
1 Set Condition Register Field 0 or Field 1 as described in Section 2.3.1, “Condition Register” on page 34.
Formats: A, M, MD, MDS, X, XFL, XO, XS, Z22, Z23

RIC (12:13)
Field used to specify what types of entries to invalidate for tlbie[].
Formats: X

RM (19:20)
Immediate operand field used to specify new binary floating-point rounding mode.
Formats: X

RMC (21:22)
Immediate field used for DFP rounding mode control.
Formats: Z23
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>RO (31)</td>
<td>Round to Odd override</td>
<td>X</td>
</tr>
<tr>
<td>RS (6:10)</td>
<td>Field used to specify a GPR to be used as a source.</td>
<td>D, DS, M, MD, MDS, X, XFX, XS</td>
</tr>
<tr>
<td>RSp (6:10)</td>
<td>Field used to specify an even/odd pair of GPRs to be concatenated and used as a source.</td>
<td>DS, X</td>
</tr>
<tr>
<td>RT (6:10)</td>
<td>Field used to specify a GPR to be used as a target.</td>
<td>A, D, DQE, DS, DX, VA, VX, X, XFX, XO, XX2</td>
</tr>
<tr>
<td>RTp (6:10)</td>
<td>Field used to specify an even/odd pair of GPRs to be concatenated and used as a target.</td>
<td>DQ, X</td>
</tr>
<tr>
<td>S (11)</td>
<td>Immediate field that specifies signed versus unsigned conversion.</td>
<td>X</td>
</tr>
<tr>
<td>S (20)</td>
<td>Immediate field that specifies whether or not the <em>rfebb</em> instruction re-enables event-based branches.</td>
<td>XL</td>
</tr>
<tr>
<td>SC (14:15)</td>
<td>Field used by the <em>Synchronize</em> instruction to specify the kind(s) of stores that are ordered.</td>
<td>X</td>
</tr>
<tr>
<td>SH (16:20)</td>
<td>Field used to specify a shift amount.</td>
<td>M, X</td>
</tr>
<tr>
<td>SH (16:21)</td>
<td>Field used to specify a shift amount.</td>
<td>Z22</td>
</tr>
<tr>
<td>sh (30,16:20)</td>
<td>Fields that are concatenated to specify a shift amount.</td>
<td>MD, XS</td>
</tr>
<tr>
<td>SHB (22:25)</td>
<td>Field used to specify a shift amount in bytes.</td>
<td>VA</td>
</tr>
<tr>
<td>SHW (22:23)</td>
<td>Field used to specify a shift amount in words.</td>
<td>XX3</td>
</tr>
<tr>
<td>SI (16:20)</td>
<td>Immediate field used to specify a 5-bit signed integer.</td>
<td>X</td>
</tr>
<tr>
<td>SI (16:31)</td>
<td>Immediate field used to specify a 16-bit signed integer.</td>
<td>D</td>
</tr>
<tr>
<td>SIM (11:15)</td>
<td>Immediate field used to specify a 5-bit signed integer.</td>
<td>VX</td>
</tr>
<tr>
<td>SP (11:12)</td>
<td>Immediate field that specifies signed versus unsigned conversion.</td>
<td>X</td>
</tr>
<tr>
<td>SPR (11:20)</td>
<td>Field used to specify a Special Purpose Register for the <em>mtspr</em> and <em>mfspr</em> instructions.</td>
<td>X</td>
</tr>
<tr>
<td>SR (12:15)</td>
<td>Field used by the <em>Segment Register Manipulation</em> instructions (see Book III).</td>
<td>X</td>
</tr>
<tr>
<td>SX,S (28,6:10)</td>
<td>Fields SX and S are concatenated to specify a VSR to be used as a source.</td>
<td>DQ</td>
</tr>
<tr>
<td>SX,S (31,6:10)</td>
<td>Fields SX and S are concatenated to specify a VSR to be used as a source.</td>
<td>X</td>
</tr>
<tr>
<td>TBR (11:20)</td>
<td>Field used by the <em>Move From Time Base</em> instruction (see Section 5.1 of Book II).</td>
<td>X</td>
</tr>
<tr>
<td>TE (11:15)</td>
<td>Immediate field that specifies a DFP exponent.</td>
<td>Z23</td>
</tr>
<tr>
<td>TH (6:10)</td>
<td>Field used by the data stream variant of the <em>dcbt</em> and <em>dcbtst</em> instructions (see Section 4.3.2 of Book II).</td>
<td>X</td>
</tr>
</tbody>
</table>
TO (6:10)
Field used to specify the conditions on which to
trap. The encoding is described in
Section 3.3.10.1, “Character-Type Compare
Instructions” on page 94.
Formats: TX, X

TX,T (28,6:10)
Fields that are concatenated to specify a VSR to
be used as either a target.
Formats: DQ

TX,T (31,6:10)
Fields that are concatenated to specify a VSR to
be used as either a target or a source.
Formats: X, XX2, XX3, XX4

U (16:19)
Immediate field used as the data to be placed into
a field in the FPSCR.
Formats: X

UI (16:20)
Immediate field used to specify a 5-bit unsigned
integer.
Formats: TX

UI (16:31)
Immediate field used to specify a 16-bit unsigned
integer.
Formats: D

UIM (11:15)
Immediate field used to specify a 5-bit unsigned
integer.
Formats: VX, X

UIM (12:15)
Immediate field used to specify a 4-bit unsigned
integer.
Formats: VX, XX2

UIM (13:15)
Immediate field used to specify a 3-bit unsigned
integer.
Formats: VX

UIM (14:15)
Immediate field used to specify a 2-bit unsigned
integer.
Formats: VX, XX2

VRA (11:15)
Field used to specify a VR to be used as a source.
Formats: VA, VC, VX

VRB (16:20)
Field used to specify a VR to be used as a source.
Formats: VA, VC, VX

VRC (21:25)
Field used to specify a VR to be used as a source.
Formats: VA

VRS (6:10)
Field used to specify a VR to be used as a source.
Formats: DS, X

VRT (6:10)
Field used to specify a VR to be used as a target.
Formats: DS, VA, VC, VX, X

W (15)
Field used by the mtfsfi and mtfsf instructions to
specify the target word in the FPSCR.
Formats: X, XFL

WC (9:10)
Field used to specify the condition or conditions
that cause instruction execution to resume after
executing a wait instruction (see Section 4.6.4 of
Book II).
Formats: X

XBI (21:24)
Field used to specify a bit in the XER.
Formats: MDS, MDS, TX

XO (21,23:31)
Extended opcode field.
Formats: VX

XO (21:24,26:28)
Extended opcode field.
Formats: XX2

XO (21:24:28)
Extended opcode field.
Formats: XX3

XO (21:28)
Extended opcode field.
Formats: XX3

XO (21:29)
Extended opcode field.
Formats: XS, XX2

XO (21:30)
Extended opcode field.
Formats: X, XFL, XFX, XL
1.6.3 Instruction Prefix Formats

Prefixed instructions consist of a 4-byte prefix followed by a 4-byte suffix. The prefix formats are specified below. The suffix formats share the same formats as word instructions, as specified in Section 1.6.1 on page 12.

Bits 0:5 of all prefixes are assigned the primary opcode value \(0b000001\). \(0b000001\) is not available for use as a primary opcode for either word instructions or suffixes of prefixed instructions.

Prefix bits 6:7 are used to identify one of four prefix formats. When bit 6 is set to 0 (prefix types 00 and 01), the suffix is not a defined word instruction (i.e., requires the prefix to identify the alternate opcode space the suffix is assigned to as well as additional or extended operand and/or control fields); when bit 6 is set to 1 (prefix types 10 and 11), the prefix is modifying the behavior of a defined word instruction in the suffix.

1.6.3.1 Type 00 Prefix – Eight-Byte Load/Store Instructions

The Type 00 prefix format provides a one-bit subtype (ST) field to specify the subformat employed by the prefix. The subformats are defined as follows.

**ST=0: Eight-Byte Load/Store Form (8LS)**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>ST</th>
<th>R</th>
<th>IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

ST=1: Reserved
1.6.3.2 Type 01 Prefix – Eight-Byte Register-to-Register Instructions

The Type 01 prefix format provides a four-bit subtype (ST) field to specify the subformat employed by the prefix. The subformats are defined as follows.

ST=0b0000: Eight-Byte Register-to-Register Form (8RR)

ST=0b0001-0b1000: Reserved

ST=0b1000-0b1001: Reserved

1.6.3.3 Type 10 - Modified Load/Store Instructions

The Type 10 prefix format provides a one-bit subtype (ST) field to specify the subformat employed by the prefix. The subformats are defined as follows.

ST=0: Modified Load/Store Form (MLS)

ST=1: Reserved

1.6.3.4 Type 11 - Modified Register-to-Register Instructions

The Type 11 prefix format provides a four-bit subtype (ST) field to specify the subformat employed by the prefix. The subformats are defined as follows.

ST=0b0000: Modified Register to Register Form (MRR)

ST=0b0001-0b1000: Reserved

ST=0b1000-0b1011: Reserved

1.6.4 Instruction Prefix Fields

IE (14:31)

18-bit immediate field that is concatenated with the D field in the suffix to extend the displacement value at the high-order end. This field is reserved when this prefix precedes instructions of other formats.

Alternate field names: d0, si0, imm18

Formats: 8LS, MLS

imm0 (16:31)

16-bit immediate field that is concatenated with the 16-bit immediate field in the suffix to create a 32-bit value.

Formats: 8RR

IMM (24:31)

8-bit immediate field used as control operand.

Formats: 8RR

PMSK (16:23)

Immediate field used to specify product mask for VSX Vector GER instructions.

Formats: MMIRR

PMSK (16:19)

Immediate field used to specify product mask for VSX Vector GER instructions.

Formats: MMIRR

PMSK (16:17)

Immediate field used to specify product mask for VSX Vector GER instructions.

Formats: MMIRR

- *pnop* (See Section 3.3.19, "Prefixed No-Operation Instruction" on page 130)
R (11)
Field used to specify whether the effective address of the storage operand is computed relative to the address of the instruction (CIA).

| 000 | Effective address is not computed relative to CIA |
| 001 | Effective address is computed relative to CIA |

Formats: 8LS, MLS

UIM (29:31)
3-bit immediate field used as control operand.

Formats: 8RR

XMSK (24:27)
Field used to specify ACC row mask for VSX Vector GER instructions.

Formats: MMIRR

YMSK (28:31)
Field used to specify ACC column mask for VSX Vector GER instructions.

Formats: MMIRR

1.7 Classes of Instructions

An instruction falls into exactly one of the following three classes:

- Defined
- Illegal
- Reserved

The class is determined by examining the opcode, and the extended opcode if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction or a reserved instruction, the instruction is illegal.

1.7.1 Defined Instruction Class

This class of instructions contains all the instructions defined in this document.

A defined instruction can have preferred and/or invalid forms, as described in Section 1.8.1, “Preferred Instruction Forms” and Section 1.8.2, “Invalid Instruction Forms”.

1.7.2 Illegal Instruction Class

This class of instructions contains the set of instructions described in Appendix B of Book Appendices. Illegal instructions are available for future extensions of the Power ISA; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

Any attempt to execute an illegal instruction will cause the system illegal instruction error handler to be invoked and will have no other effect.

An instruction consisting entirely of binary 0s is guaranteed always to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized storage will result in the invocation of the system illegal instruction error handler.

1.7.3 Reserved Instruction Class

This class of instructions contains the set of instructions described in Appendix C of Book Appendices.

Reserved instructions are allocated to specific purposes that are outside the scope of the Power ISA.

Any attempt to execute a reserved instruction will:
- perform the actions described by the implementation if the instruction is implemented; or
- cause the system illegal instruction error handler to be invoked if the instruction is not implemented.

1.8 Forms of Defined Instructions

1.8.1 Preferred Instruction Forms

Some of the defined instructions have preferred forms. For such an instruction, the preferred form will execute in an efficient manner, but any other form may take significantly longer to execute than the preferred form.

Instructions having preferred forms are:
- the Condition Register Logical instructions
- the Load Quadword instruction
- the Move Assist instructions
- the Or Immediate instruction (preferred form of no-op)
- the Move To Condition Register Fields instruction

1.8.2 Invalid Instruction Forms

Some of the defined instructions can be coded in a form that is invalid. An instruction form is invalid if one or more fields of the instruction, excluding the opcode field(s), are coded incorrectly in a manner that can be deduced by examining only the instruction encoding.

In general, any attempt to execute an invalid form of an instruction will either cause the system illegal instruction error handler to be invoked or yield boundedly undefined results. Exceptions to this rule are stated in the instruction descriptions.
Some instruction forms are invalid because the instruction contains a reserved value in a defined field (see Section 1.3.3 on page 5); these invalid forms are not discussed further. All other invalid forms are identified in the instruction descriptions.

References to instructions elsewhere in this document assume the instruction form is not invalid, unless otherwise stated or obvious from context.

### Assembler Note

Assemblers should report uses of invalid instruction forms as errors.

#### 1.8.3 Reserved-no-op Instructions

Reserved-no-op instructions include the following extended opcodes under primary opcode 31: 530, 562, 594, 626, 658, 690, 722, and 754.

Reserved-no-op instructions are provided in the architecture to anticipate the eventual adoption of performance hint instructions to the architecture. For these instructions, which cause no visible change to architected state, employing a reserved-no-op opcode will allow software to use this new capability on new implementations that support it while remaining compatible with existing implementations that may not support the new function.

When a reserved-no-op instruction is executed, no operation is performed.

Reserved-no-op instructions are not assigned instruction names or mnemonics. There are no individual descriptions of reserved-no-op instructions in this document.

#### 1.9 Exceptions

There are two kinds of exception, those caused directly by the execution of an instruction and those caused by an asynchronous event. In either case, the exception may cause one of several components of the system software to be invoked.

The exceptions that can be caused directly by the execution of an instruction include the following:

- an attempt to execute an illegal instruction, or an attempt by an application program to execute a “privileged” instruction (see Book III) (system illegal instruction error handler or system privileged instruction error handler)
- the execution of a defined instruction using an invalid form (system illegal instruction error handler or system privileged instruction error handler)
- an attempt to execute an instruction that is not provided by the implementation (system illegal instruction error handler)
- an attempt to execute a prefixed instruction that crosses a 64-byte address boundary. (system alignment error handler)
- an attempt to access a storage location that is unavailable (system instruction storage error handler or system data storage error handler)
- an attempt to access storage with an effective address alignment that is invalid for the instruction (system alignment error handler)
- the execution of a System Call or System Call Vectorized instruction (system service program)
- the execution of a Trap instruction that traps (system trap handler)
- the execution of a floating-point instruction that causes a floating-point enabled exception to exist (system floating-point enabled exception error handler)

The exceptions that can be caused by an asynchronous event are described in Book III.

The invocation of the system error handler is precise, except that if one of the imprecise modes for invoking the system floating-point enabled exception error handler is in effect (see page 141), then the invocation of the system floating-point enabled exception error handler may also be imprecise. When the system error handler is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the system error handler, has not yet occurred).

Additional information about exception handling can be found in Book III.

#### 1.10 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a Storage Access or Branch instruction (or certain other instructions described in Book II and Book III), or when it fetches the next sequential instruction.

Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

The byte ordering (Big-Endian or Little-Endian) for a storage access is specified by the operating system. This byte ordering is also referred to as the Endian mode and it applies to both data accesses and instruction fetches. The Endian mode is specified by the LE mode bit (see Section 4.2.1 of Book III), which applies to all of storage.
1.10.1 Storage Operands

A storage operand may be a byte, a halfword, a word, a doubleword, a quadword, an octword, or, for the Load/Store Multiple, Move Assist, and Load/Store VSX Vector with Length [Left-justified] instructions, a sequence of bytes (Move Assist and Load/Store VSX Vector with Length [Left-justified]) or words (Load/Store Multiple). The address of a storage operand is the address of its first byte (i.e., of its lowest-numbered byte). An instruction for which the storage operand is a byte is said to cause a byte access, and similarly for halfword, word, doubleword, quadword, and octword.

The length of the storage operand is the number of bytes (of the storage operand) that the instruction would access in the absence of invocations of the system error handler. The length is generally implied by the name of the instruction (equivalently, by the opcode, and extended opcode if any). For example, the length of the storage operand of a Load Word and Zero, Load Floating-Point Single, and Load Vector Element Word instruction is four bytes (one word), the length of a Store Quadword, Store Floating-Point Double Pair, and Store VSX Vector Word*4 instruction is 16 bytes (one quadword), and the length of a Load VSX Vector Paired instruction is 32 bytes (one octword). The only exceptions are the Load/Store Multiple and Move Assist instructions, for which the length of the storage operand is implied by the identity of the specified source or target register (Load/Store Multiple), or by an immediate field in the instruction or the contents of a field in the XER (Move Assist), as well as by the name of the instruction. For example, the length of the storage operand of a Load Multiple Word instruction for which the specified target register is GPR 20 is 48 bytes ((32-20)x4), and the length of the storage operand of a Load String Word Immediate instruction for which the immediate field contains the number 20 is 20 bytes.

The storage operand of a Load or Store instruction other than a Load/Store Multiple or Move Assist instruction is said to be aligned if the address of the storage operand is an integral multiple of the storage operand length; otherwise it is said to be unaligned. See the following table. (The storage operand of a Load/Store Multiple or Move Assist instruction is neither said to be aligned nor said to be unaligned. Its alignment properties are described, when necessary, using terms such as “word-aligned”, which are defined below.)

<table>
<thead>
<tr>
<th>Operand</th>
<th>Length</th>
<th>Addr&lt;6:4&gt; if aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>8 bits</td>
<td>x x x x x x</td>
</tr>
<tr>
<td>Halfword</td>
<td>2 bytes</td>
<td>x x x x x 0</td>
</tr>
<tr>
<td>Word</td>
<td>4 bytes</td>
<td>x x 0 0 0 0</td>
</tr>
<tr>
<td>Doubleword</td>
<td>8 bytes</td>
<td>x 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Quadword</td>
<td>16 bytes</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Octword</td>
<td>32 bytes</td>
<td>0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Note: An “x” in an address bit position indicates that the bit can be 0 or 1 independent of the contents of other bits in the address.

The concept of alignment is also applied more generally, to any datum in storage.

A datum having length that is an integral power of 2 is said to be aligned if its address is an integral multiple of its length.

A datum of any length is said to be halfword-aligned (or aligned at a halfword boundary) if its address is an integral multiple of 2, word-aligned (or aligned at a word boundary) if its address is an integral multiple of 4, etc. (All data in storage is byte-aligned.)

The concept of alignment can also be applied to data in registers, with the “address” of the datum interpreted as the byte number of the datum in the register. E.g., a word element (4 bytes) in a Vector Register is said to be aligned if its byte number is an integral multiple of 4.

Programming Note

The technical literature sometimes uses the term “naturally aligned” to mean “aligned.” Versions of the architecture that precede Version 2.07 also used “naturally aligned” as defined above. The term was dropped from the architecture in Version 2.07 because it seemed to mean different things to different readers and is not needed.

Some instructions require their storage operands to have certain alignments. In addition, alignment may affect performance. In general, the best performance is obtained when storage operands are aligned.

When a storage operand of length N bytes starting at effective address EA is copied between storage and a register that is R bytes long (i.e., the register contains bytes numbered from 0, most significant, through R-1, least significant), the bytes of the operand are placed into the register or into storage in a manner that depends on the byte ordering for the storage access as shown in Figure 28, unless otherwise specified in the instruction description.
Chapter 1. Introduction

**Figure 28. Storage operands and byte ordering**

Figure 29 shows an example of a C language structure `s` containing an assortment of scalars and one character string. The value assumed to be in each structure element is shown in hex in the C comments; these values are used below to show how the bytes making up each structure element are mapped into storage. It is assumed that structure `s` is compiled for 32-bit mode or for a 32-bit implementation. (This affects the length of the pointer to `c`.)

C structure mapping rules permit the use of padding (skipped bytes) in order to align the scalars on desirable boundaries. Figures 30 and 31 show each scalar as aligned. This alignment introduces padding of four bytes between `a` and `b`, one byte between `d` and `e`, and two bytes between `e` and `f`. The same amount of padding is present for both Big-Endian and Little-Endian mappings.

The Big-Endian mapping of structure `s` is shown in Figure 30. Addresses are shown in hex at the left of each doubleword, and in small figures below each byte. The contents of each byte, as indicated in the C example in Figure 29, are shown in hex (as characters for the elements of the string).

The Little-Endian mapping of structure `s` is shown in Figure 31. Doublewords are shown laid out from right to left, which is the common way of showing storage maps for processors that implement only Little-Endian byte ordering.

---

**Big-Endian Byte Ordering**

<table>
<thead>
<tr>
<th>Load</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>do i = 0 to N-1:</code></td>
<td><code>RT[(B.N)+i] ← MEM[EA+i,1]</code></td>
</tr>
</tbody>
</table>

**Little-Endian Byte Ordering**

<table>
<thead>
<tr>
<th>Load</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>do i = 0 to N-1:</code></td>
<td><code>RT[(B.1)+i] ← MEM[EA+i,1]</code></td>
</tr>
</tbody>
</table>

**Notes:**
1. In this table, subscripts refer to bytes in a register.
2. This table does not apply to the `lvebx`, `lvehx`, `lvex`, `stvebx`, `stvehx`, and `stvex` instructions.

---

**Figure 29. C structure 's', showing values of elements**

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>00</strong></td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td><strong>08</strong></td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td><strong>0B</strong></td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
</tr>
<tr>
<td><strong>10</strong></td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td><strong>18</strong></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td><strong>20</strong></td>
<td>51</td>
<td>52</td>
<td>5A</td>
<td>5B</td>
<td>5C</td>
<td>5D</td>
<td>5E</td>
<td>5F</td>
</tr>
</tbody>
</table>

---

**Figure 30. Big-Endian mapping of structure 's'**

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>00</strong></td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td><strong>08</strong></td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td><strong>0B</strong></td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
</tr>
<tr>
<td><strong>10</strong></td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td><strong>18</strong></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td><strong>20</strong></td>
<td>51</td>
<td>52</td>
<td>5A</td>
<td>5B</td>
<td>5C</td>
<td>5D</td>
<td>5E</td>
<td>5F</td>
</tr>
</tbody>
</table>

---

**Figure 31. Little-Endian mapping of structure 's'**

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>00</strong></td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td><strong>08</strong></td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td><strong>0B</strong></td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
</tr>
<tr>
<td><strong>10</strong></td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td><strong>18</strong></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td><strong>20</strong></td>
<td>51</td>
<td>52</td>
<td>5A</td>
<td>5B</td>
<td>5C</td>
<td>5D</td>
<td>5E</td>
<td>5F</td>
</tr>
</tbody>
</table>

---

**Big-Endian Byte Ordering**

<table>
<thead>
<tr>
<th>Load</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>do i = 0 to N-1:</code></td>
<td><code>RT[(B.N)+i] ← MEM[EA+i,1]</code></td>
</tr>
</tbody>
</table>

**Little-Endian Byte Ordering**

<table>
<thead>
<tr>
<th>Load</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>do i = 0 to N-1:</code></td>
<td><code>RT[(B.1)+i] ← MEM[EA+i,1]</code></td>
</tr>
</tbody>
</table>

**Notes:**
1. In this table, subscripts refer to bytes in a register.
2. This table does not apply to the `lvebx`, `lvehx`, `lvex`, `stvebx`, `stvehx`, and `stvex` instructions.
1.10.2 Instruction Fetches

Instructions are encoded in either four or eight bytes and are word-aligned. For purposes of byte ordering, prefixed instructions are treated as if they are two independent four-byte instructions, with the prefix preceding the suffix in storage regardless of the Endian mode.

When an instruction starting at effective address EA is fetched from storage, the relative order of the bytes within each word of the instruction image depends on the byte ordering for the storage access as shown in Figure 32.

The Big-Endian mapping of program p is shown in Figure 34 (assuming the program starts at address 0).

The Little-Endian mapping of program p is shown in Figure 35.

Figure 32. Instructions and byte ordering

Figure 33 shows an example of a small assembly language program p. In the program, prefixed instruction 1 is doubleword-aligned and prefixed instruction 2 is word-aligned.

Figure 33. Assembly language program ‘p’
Programming Note

The terms Big-Endian and Little-Endian come from Part I, Chapter 4, of Jonathan Swift’s Gulliver’s Travels. Here is the complete passage, from the edition printed in 1734 by George Faulkner in Dublin.

... our Histories of six Thousand Moons make no Mention of any other Regions, than the two great Empires of Lilliput and Blefuscu. Which two mighty Powers have, as I was going to tell you, been engaged in a most obstinate War for six and thirty Moons past. It began upon the following Occasion. It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majesty’s Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were constantly fomented by the Monarchs of Lilliput and Blefuscu; and when they were quelled, the Exiles always fled for Refuge to that Empire. It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the Books of the Big-Endians have been long forbidden, and the whole Party rendered incapable by Law of holding Employments. During the Course of these Troubles, the Emperors of Blefuscu did frequently expostulate by their Ambassadors, accusing us of making a Schism in Religion, by offending against a fundamental Doctrine of our great Prophet Lustrog, in the fifty-fourth Chapter of the Brundrecal, (which is their Alcoran.) This, however, is thought to be a mere Strain upon the text: For the Words are these: That all true Believers shall break their Eggs at the convenient End: and which is the convenient End, seems, in my humble Opinion, to be left to every Man’s Conscience, or at least in the Power of the chief Magistrate to determine. Now the Big-Endian Exiles have found so much Credit in the Emperor of Blefuscu’s Court; and so much private Assistance and Encouragement from their Party here at home, that a bloody War has been carried on between the two Empires for six and thirty Moons with various Success; during which Time we have lost Forty Capital Ships, and a much greater Number of smaller Vessels, together with thirty thousand of our best Sea-men and Soldiers; and the Damage received by the Enemy is reckoned to be somewhat greater than ours. However, they have now equipped a numerous Fleet, and are just preparing to make a Descent upon us: and his Imperial Majesty, placing great Confidence in your Valour and Strength, hath commanded me to lay this Account of his Affairs before you.

1.10.3 Effective Address Calculation

An effective address is computed by the processor when executing a Storage Access or Branch instruction (or certain other instructions described in Book II and Book III) when fetching the next sequential instruction, or when invoking a system error handler. The following provides an overview of this process. More detail is provided in the individual instruction descriptions.

Effective address calculations, for both data and instruction accesses, use 64-bit two’s complement addition. All 64 bits of each address component participate in the calculation regardless of mode (32-bit or 64-bit). In this computation one operand is an address (which is by definition an unsigned number) and the second is a signed offset. Carries out of the most significant bit are ignored.

In 64-bit mode, the entire 64-bit result comprises the 64-bit effective address. The effective address arithmetic wraps around from the maximum address, \(2^{64} - 1\), to address 0, except that if the current instruction is a word instruction at effective address \(2^{64} - 4\) or a prefixed instruction at effective address \(2^{64} - 8\), the effective address of the next sequential instruction is undefined, and if the current instruction is a prefixed instruction at effective address \(2^{64} - 4\), the effective address of the suffix is undefined.

In 32-bit mode, the low-order 32 bits of the 64-bit result, preceded by 32 0 bits, comprise the 64-bit effective address for the purpose of addressing storage, except that if the current instruction is a word instruction at effective address \(2^{32} - 4\) or a prefixed instruction at effective address \(2^{32} - 8\), the 64-bit effective address of the next sequential instruction is undefined, and if the current instruction is a prefixed instruction at effective address \(2^{32} - 4\), the effective address of the suffix is undefined. Thus, as used to address storage, the effective address arithmetic appears to wrap around from
the maximum address $^{232-1}$, to address 0, except when the resulting 64-bit effective address is undefined as just described. When an effective address is placed into a register by an instruction or event, the value placed into the register is as follows.

- **Register RA** when set by Load with Update and Store with Update instructions: the entire 64-bit result.

- **All other cases** (e.g., the Link Register when set by Branch instructions having LK=1, Special Purpose Registers when set to an effective address by invocation of a system error handler): the low-order 32 bits of the 64-bit result preceded by 32 0 bits, except that if the intended effective address is that of the NIA of either a word instruction at effective address $^{232-4}$, or a prefixed instruction at effective address $^{232-8}$, the value placed into the register is undefined.

RA is a field in the instruction which specifies an address component in the computation of an effective address. A zero in the RA field indicates the absence of the corresponding address component. A value of zero is substituted for the absent component of the effective address computation. This substitution is shown in the instruction descriptions as (RA=0).

Effective addresses are computed as follows. In the descriptions below, it should be understood that “the contents of a GPR” refers to the entire 64-bit contents, independent of mode, but that in 32-bit mode only bits 32:63 of the 64-bit result of the computation are used to address storage.

- **With X-form instructions**, in computing the effective address of a data element, the contents of the GPR designated by RB (or the value zero for ldat, lswi, lwat, lxvl, lxvll, stdat, stswi, stwat, stxvl, and stxvll) are added to the contents of the GPR designated by RA or to zero if RA=0 or RA is not used in forming the EA.

- **With X-form instructions** that are preceded by an MLS-form or MMLS-form prefix with the R bit set to 1 (see xref to Section 1.6.2), in computing the effective address of the data element, the contents of the GPR designated by RB are added to the CIA and RA is not used in forming the EA.

- **With DQ-form instructions**, the 16-bit D field is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.

- **With prefixed instructions having a D-form suffix** and an MLS-form or 8LS-form prefix, the 16-bit D field is concatenated on the left with the 18-bit IE field in the prefix, and the concatenation is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0. If the R bit in the prefix is set to 0, or is added to the CIA if the R bit in the prefix is set to 1.

- **With prefixed instructions** having a D-form suffix and an MMLS-form or 8MLS-form prefix, the 16-bit D field is concatenated on the left with the 12-bit IE field in the prefix, and the concatenation is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0 if the R bit in the prefix is set to 0, or is added to the CIA if the R bit in the prefix is set to 1.

- **With DS-form instructions**, the 14-bit DS field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.

- **With prefixed instructions** having a DS-form suffix and an MLS-form or 8LS-form prefix, the 14-bit D field is concatenated on the right with 0b00 and is concatenated on the left with the 12-bit IE field in the prefix, and the concatenation is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0 if the R bit in the prefix is set to 0, or is added to the CIA if the R bit in the prefix is set to 1.

- **With prefixed instructions** having a DS-form suffix and an MMLS-form or 8MLS-form prefix, the 14-bit D field is concatenated on the right with 0b00 and is concatenated on the left with the 12-bit IE field in the prefix, and the concatenation is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0 if the R bit in the prefix is set to 0, or is added to the CIA if the R bit in the prefix is set to 1.

- **With prefixed instructions** having a DS-form suffix and an MLS-form or 8LS-form prefix, the 12-bit DQ field is concatenated on the right with 0b0000 and sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.

- **With prefixed instructions** having a DQ-form suffix and an MLS-form or 8LS-form prefix, the 12-bit DQ field is concatenated on the right with 0b0000 and is concatenated on the left with the 18-bit IE field in the prefix, and the concatenation is sign-extended to form a 64-bit address component. In computing
the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0 if the R bit in the prefix is set to 0, or is added to the CIA if the R bit in the prefix is set to 1.

With prefixed instructions having a DQ-form suffix and an MMLS-form or 8MLS-form prefix, the 12-bit DQ field is concatenated on the right with 0b0000 and is concatenated on the left with the 12-bit IE field in the prefix, and the concatenation is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0 if the R bit in the prefix is set to 0, or is added to the CIA if the R bit in the prefix is set to 1.

- With I-form Branch instructions, the 24-bit LI field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the Branch instruction to form the effective address of the target instruction. If AA=1, this address component is the effective address of the target instruction.

- With B-form Branch instructions, the 14-bit BD field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the Branch instruction to form the effective address of the target instruction. If AA=1, this address component is the effective address of the target instruction.

- With XL-form Branch instructions, bits 0:61 of the Link Register or the Count Register are concatenated on the right with 0b00 to form the effective address of the target instruction.

- With sequential instruction fetching, if the current instruction is a word instruction, the value 4 is added to the address of the current instruction to form the effective address of the next instruction, and if the current instruction is a prefixed instruction, the value 8 is added to the address of the current instruction to form the effective address of the next instruction, except that if the current instruction is at the maximum instruction effective address for the mode (for a word instruction, $2^{64} - 4$ in 64-bit mode and $2^{32} - 4$ in 32-bit mode; for a prefixed instruction, $2^{64} - 8$ in 64-bit mode and $2^{32} - 8$ in 32-bit mode) the effective address of the next sequential instruction is undefined.

If the size of the operand of a Storage Access instruction is more than one byte, the effective address for each byte after the first is computed by adding 1 to the effective address of the preceding byte.
Chapter 2. Branch Facility

2.1 Branch Facility Overview

This chapter describes the registers and instructions that make up the Branch Facility.

2.2 Instruction Execution Order

In general, instructions appear to execute sequentially, in the order in which they appear in storage. The exceptions to this rule are listed below.

- **Branch** instructions for which the branch is taken cause execution to continue at the target address specified by the Branch instruction.

- **Trap** instructions for which the trap conditions are satisfied, and **System Call** and **System Call Vectored** instructions, cause the appropriate system handler to be invoked.

- Event-based exceptions can cause the event-based branch handler to be invoked, as described in Chapter 6 of Book II.

- Exceptions can cause the system error handler to be invoked, as described in Section 1.9, “Exceptions” on page 25.

- Returning from a system service program, system trap handler, or system error handler causes execution to continue at a specified address.

The model of program execution in which the processor appears to execute one instruction at a time, completing each instruction before beginning to execute the next instruction is called the “sequential execution model”. In general, the processor obeys the sequential execution model. For the instructions and facilities defined in this Book, the only exceptions to this rule are the following.

- A floating-point exception occurs when the processor is running in one of the Imprecise floating-point exception modes (see Section 4.4). The instruction that causes the exception need not complete before the next instruction begins execution, with respect to setting exception bits and (if the exception is enabled) invoking the system error handler.

- A **Store** instruction modifies one or more bytes in an area of storage that contains instructions that will subsequently be executed. Before an instruction in that area of storage is executed, software synchronization is required to ensure that the instructions executed are consistent with the results produced by the Store instruction.

---

**Programming Note**

This software synchronization will generally be provided by system library programs (see Section 1.8 of Book II). Application programs should call the appropriate system library program before attempting to execute modified instructions.
2.3 Branch Facility Registers

2.3.1 Condition Register

The Condition Register (CR) is a 32-bit register which reflects the result of certain operations, and provides a mechanism for testing (and branching).

![Condition Register Diagram]

The bits in the Condition Register are grouped into eight 4-bit fields, named CR Field 0 (CR0), ..., CR Field 7 (CR7), which are set in one of the following ways.

- Specified fields of the CR can be set by a move to the CR from a GPR (mtcrf, mtocrf).
- A specified field of the CR can be set by a move to the CR from another CR field (mcrf), from OV, CA, OV32, and CA32 (mcrxr), or from the FPSCR (mcrfs).
- CR Field 0 can be set as the implicit result of a fixed-point instruction.
- CR Field 1 can be set as the implicit result of a floating-point instruction.
- CR Field 1 can be set as the implicit result of a decimal floating-point instruction.
- CR Field 6 can be set as the implicit result of a vector instruction.
- A specified CR field can be set as the result of a Compare instruction.

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits.

For all fixed-point instructions in which Rc=1, and for addic, andi, and andis, the first three bits of CR Field 0 (bits 32:34 of the Condition Register) are set by signed comparison of the result to zero, and the fourth bit of CR Field 0 (bit 35 of the Condition Register) is copied from the SO field of the XER. "Result" here refers to the entire 64-bit value placed into the target register in 64-bit mode, and to bits 32:63 of the 64-bit value placed into the target register in 32-bit mode.

```plaintext
if (64-bit mode)
    then M ← 0
else M ← 32
if (target_register)M,63 < 0 then c ← 0b100
else if (target_register)M,63 > 0 then c ← 0b010
else c ← 0b001
CR0 ← c || XERSO
```

If any portion of the result is undefined, then the value placed into the first three bits of CR Field 0 is undefined.

The bits of CR Field 0 are interpreted as follows.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Negative (LT)</td>
</tr>
<tr>
<td>1</td>
<td>Positive (GT)</td>
</tr>
<tr>
<td>2</td>
<td>Zero (EQ)</td>
</tr>
<tr>
<td>3</td>
<td>Summary Overflow (SO)</td>
</tr>
</tbody>
</table>

This is a copy of the contents of XERSO at the completion of the instruction.

The `paste` instruction (see Section 4.4, “Copy-Paste Facility”, in Book II) and the stbcx, sthcx, stwcx, stdcx, and stqcx instructions (see Section 4.6.2, “Load And Reserve and Store Conditional Instructions”, in Book II) also set CR Field 0.

For all floating-point instructions in which Rc=1, CR Field 1 (bits 36:39 of the Condition Register) is set to the Floating-Point exception status, copied from bits 32:35 of the Floating-Point Status and Control Register. This occurs regardless of whether any exceptions are enabled, and regardless of whether the writing of the result is suppressed (see Section 4.4, “Floating-Point Exceptions” on page 140). These bits are interpreted as follows.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Floating-Point Exception Summary (FX)</td>
</tr>
<tr>
<td></td>
<td>This is a copy of the contents of FPSCRFX at</td>
</tr>
<tr>
<td></td>
<td>the completion of the instruction.</td>
</tr>
<tr>
<td>33</td>
<td>Floating-Point Enabled Exception Summary (FEX)</td>
</tr>
<tr>
<td></td>
<td>This is a copy of the contents of FPSCRFEX at</td>
</tr>
<tr>
<td></td>
<td>the completion of the instruction.</td>
</tr>
<tr>
<td>34</td>
<td>Floating-Point Invalid Operation Exception</td>
</tr>
<tr>
<td></td>
<td>Summary (VX)</td>
</tr>
<tr>
<td></td>
<td>This is a copy of the contents of FPSCRVX at</td>
</tr>
<tr>
<td></td>
<td>the completion of the instruction.</td>
</tr>
<tr>
<td>35</td>
<td>Floating-Point Overflow Exception (OX)</td>
</tr>
<tr>
<td></td>
<td>This is a copy of the contents of FPSCRFX at</td>
</tr>
<tr>
<td></td>
<td>the completion of the instruction.</td>
</tr>
</tbody>
</table>

For Compare instructions, a specified CR field is set to reflect the result of the comparison. The bits of the specified CR field are interpreted as follows. A complete description of how the bits are set is given in the instruction descriptions in Section 3.3.10, “Fixed-Point Compare Instructions” on page 92, and Section 4.6.8, “Floating-Point Compare Instructions” on page 179.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Less Than, Floating-Point Less Than (LT, FL)</td>
</tr>
</tbody>
</table>
For fixed-point Compare instructions, (RA) < SI or (RB) (signed comparison) or (RA) < UI or (RB) (unsigned comparison). For floating-point Compare instructions, (FRA) < (FRB).

1 **Greater Than, Floating-Point Greater Than** (GT, FG)
   For fixed-point Compare instructions, (RA) > SI or (RB) (signed comparison) or (RA) > UI or (RB) (unsigned comparison). For floating-point Compare instructions, (FRA) > (FRB).

2 **Equal, Floating-Point Equal** (EQ, FE)
   For fixed-point Compare instructions, (RA) = SI, UI, or (RB). For floating-point Compare instructions, (FRA) = (FRB).

3 **Summary Overflow, Floating-Point Unordered** (SO, FU)
   For fixed-point Compare instructions, this is a copy of the contents of XERSO at the completion of the instruction. For floating-point Compare instructions, one or both of (FRA) and (FRB) is a NaN.

The Vector Integer Compare instructions (see Section 6.9.3, ”Vector Integer Compare Instructions“) compare two Vector Registers element by element, interpreting the elements as unsigned or signed integers depending on the instruction, and set the corresponding element of the target Vector Register to all 1s if the relation being tested is true and 0s if the relation being tested is false.

If Rc=1, CR Field 6 is set to reflect the result of the comparison, as follows

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The relation is true for all element pairs (i.e., VRT is set to all 1s).</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>The relation is false for all element pairs (i.e., VRT is set to all 0s).</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

The Vector Floating-Point Compare instructions compare two Vector Registers word element by word element, interpreting the elements as single-precision floating-point numbers. With the exception of the Vector Compare Bounds Floating-Point instruction, they set the target Vector Register, and CR Field 6 if Rc=1, in the same manner as do the Vector Integer Compare instructions.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The relation is true for all element pairs (i.e., VRT is set to all 1s).</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

2 The relation is false for all element pairs (i.e., VRT is set to all 0s).
3 0

The Vector Compare Bounds Floating-Point instruction on page 429 sets CR Field 6 if Rc=1, to indicate whether the elements in VRA are within the bounds specified by the corresponding element in VRB, as explained in the instruction description. A single-precision floating-point value x is said to be ”within the bounds” specified by a single-precision floating-point value y if \(-y \leq x \leq y\).

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Set to indicate whether all four elements in VRA are within the bounds specified by the corresponding element in VRB, otherwise set to 0.</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

### 2.3.2 Link Register

The Link Register (LR) is a 64-bit register. It can be used to provide the branch target address for the Branch Conditional to Link Register instruction, and it holds the return address after Branch instructions for which LK=1 and after System Call Vectored instructions.

<table>
<thead>
<tr>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

Figure 37. Link Register

### 2.3.3 Count Register

The Count Register (CTR) is a 64-bit register. It can be used to hold a loop count that can be decremented during execution of Branch instructions that contain an appropriately coded BO field. If the value in the Count Register is 0 before being decremented, it is -1 afterward. The Count Register can also be used to provide the branch target address for the Branch Conditional to Count Register instruction. The Count Register is modified by the System Call Vectored instruction.

<table>
<thead>
<tr>
<th>CTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

Figure 38. Count Register

### 2.3.4 Target Address Register

The Target Address Register (TAR) is a 64-bit register. It can be used to provide bits 0:61 of the branch target
address for the Branch Conditional to Branch Target Address Register instruction. Bits 62:63 are ignored by the hardware but can be set and reset by software.

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>62</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 39. Target Address Register

**Programming Note**
The TAR is reserved for system software.
2.4 Branch Instructions

The sequence of instruction execution can be changed by the Branch instructions. Because all instructions are on word boundaries, bits 62 and 63 of the generated branch target address are ignored by the processor in performing the branch.

The Branch instructions compute the effective address (EA) of the target in one of the following five ways, as described in Section 1.10.3, “Effective Address Calculation” on page 29.

1. Adding a displacement to the address of the Branch instruction (Branch or Branch Conditional with AA=0).
2. Specifying an absolute address (Branch or Branch Conditional with AA=1).
3. Using the address contained in the Link Register (Branch Conditional to Link Register).
4. Using the address contained in the Count Register (Branch Conditional to Count Register).
5. Using the address contained in the Target Address Register (Branch Conditional to Target Address Register).

In all five cases, in 32-bit mode the final step in the address computation is setting the high-order 32 bits of the target address to 0.

For the first two methods, the target addresses can be computed sufficiently ahead of the Branch instruction that instructions can be prefetched along the target path. For the third through fifth methods, prefetching instructions along the target path is also possible provided the Link Register or the Count Register is loaded sufficiently ahead of the Branch instruction.

Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided (LK=1), the effective address of the instruction following the Branch instruction is placed into the Link Register after the branch target address has been computed; this is done regardless of whether the branch is taken.

For Branch Conditional instructions, the BO field specifies the conditions under which the branch is taken, as shown in Figure 40. In the figure, M=0 in 64-bit mode and M=32 in 32-bit mode.

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000z</td>
<td>Decrement the CTR, then branch if the decremented ( CTR_{M:63} \neq 0 ) and ( CR_{BI}=0 )</td>
</tr>
<tr>
<td>001z</td>
<td>Decrement the CTR, then branch if the decremented ( CTR_{M:63} = 0 ) and ( CR_{BI}=0 )</td>
</tr>
<tr>
<td>01at</td>
<td>Branch if ( CR_{BI}=0 )</td>
</tr>
<tr>
<td>0100z</td>
<td>Decrement the CTR, then branch if the decremented ( CTR_{M:63} \neq 0 ) and ( CR_{BI}=1 )</td>
</tr>
<tr>
<td>0101z</td>
<td>Decrement the CTR, then branch if the decremented ( CTR_{M:63} = 0 ) and ( CR_{BI}=1 )</td>
</tr>
<tr>
<td>011at</td>
<td>Branch if ( CR_{BI}=1 )</td>
</tr>
<tr>
<td>1a00t</td>
<td>Decrement the CTR, then branch if the decremented ( CTR_{M:63} \neq 0 )</td>
</tr>
<tr>
<td>1a01t</td>
<td>Decrement the CTR, then branch if the decremented ( CTR_{M:63} = 0 )</td>
</tr>
<tr>
<td>1z1zz</td>
<td>Branch always</td>
</tr>
</tbody>
</table>

Notes:
1. “z” denotes a bit that is ignored.
2. The “a” and “t” bits are used as described below.

Figure 40. BO field encodings

The “a” and “t” bits of the BO field can be used by software to provide a hint about whether the branch is likely to be taken or is likely not to be taken, as shown in Figure 41.

<table>
<thead>
<tr>
<th>at</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No hint is given</td>
</tr>
<tr>
<td>01</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>The branch is very likely not to be taken</td>
</tr>
<tr>
<td>11</td>
<td>The branch is very likely to be taken</td>
</tr>
</tbody>
</table>

Figure 41. “at” bit encodings

Programming Note

Many implementations have dynamic mechanisms for predicting whether a branch will be taken. Because the dynamic prediction is likely to be very accurate, and is likely to be overridden by any hint provided by the “at” bits, the “at” bits should be set to 0b00 unless the static prediction implied by at=0b10 or at=0b11 is highly likely to be correct.

For Branch Conditional to Link Register, Branch Conditional to Count Register, and Branch Conditional to Target Address Register instructions, the BH field provides...
a hint about the use of the instruction, as shown in Figure 42.

<table>
<thead>
<tr>
<th>BH</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>\textit{bclr}[i]: The instruction is a subroutine return \textit{bcctr}[i] and \textit{bctar}[i]: The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken</td>
</tr>
<tr>
<td>01</td>
<td>\textit{bcctr}[i]: The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken \textit{bctar}[i]: Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>\textit{bclr}[i], \textit{bcctr}[i], and \textit{bctar}[i]: The target address is not predictable</td>
</tr>
</tbody>
</table>

**Figure 42. BH field encodings**

**Programming Note**

The hint provided by the BH field is independent of the hint provided by the “at” bits (e.g., the BH field provides no indication of whether the branch is likely to be taken).

**Extended mnemonics for branches**

Many extended mnemonics are provided so that \textit{Branch Conditional} instructions can be coded with portions of the BO and BI fields as part of the mnemonic rather than as part of a numeric operand. Some of these are shown as examples with the Branch instructions. See Appendix C for additional extended mnemonics.

**Programming Note**

The hints provided by the “at” bits and by the BH field do not affect the results of executing the instruction.

The “z” bits should be set to 0, because they may be assigned a meaning in some future version of the architecture.
Many implementations have dynamic mechanisms for predicting the target addresses of \texttt{bclr[l]} and \texttt{bcctr[l]} instructions. These mechanisms may cache return addresses (i.e., Link Register values set by \textit{Branch} instructions for which \textit{LK}=1 and for which the branch was taken, other than the special form shown in the first example below) and recently used branch target addresses. To obtain the best performance across the widest range of implementations, the programmer should obey the following rules.

- Use \textit{Branch} instructions for which \textit{LK}=1 only as subroutine calls (including function calls, etc.), or in the special form shown in the first example below.
- Pair each subroutine call (i.e., each \textit{Branch} instruction for which \textit{LK}=1 and the branch is taken, other than the special form shown in the first example below) with a \texttt{bclr} instruction that returns from the subroutine and has BH=0b00.
- Do not use \texttt{bclrl} as a subroutine call. (Some implementations access the return address cache at most once per instruction; such implementations are likely to treat \texttt{bclrl} as a subroutine return, and not as a subroutine call.)
- For \texttt{bclr[l]} and \texttt{bcctr[l]}, use the appropriate value in the BH field.

The following are examples of programming conventions that obey these rules. In the examples, BH is assumed to contain 0b00 unless otherwise stated. In addition, the “at” bits are assumed to be coded appropriately.

Let A, B, and Glue be specific programs.

- Obtaining the address of the next instruction:
  Use the following form of \textit{Branch and Link}.
  \begin{verbatim}
  bcl 20,31,$+4
  \end{verbatim}

- Loop counts:
  Keep them in the Count Register, and use a \texttt{bc} instruction (LK=0) to decrement the count and to branch back to the beginning of the loop if the decremented count is nonzero.

- Computed goto’s, case statements, etc.:
  Use the Count Register to hold the address to branch to, and use a \texttt{bcctr} instruction (LK=0, and BH=0b11 if appropriate) to branch to the selected address.

- Direct subroutine linkage:
  Here A calls B and B returns to A. The two branches should be as follows.
  - A calls B: use a \texttt{bl} or \texttt{bcl} instruction (LK=1).
  - B returns to A: use a \texttt{bclr} instruction (LK=0) (the return address is in, or can be restored to, the Link Register).

- Indirect subroutine linkage:
  Here A calls Glue, Glue calls B, and B returns to A rather than to Glue. (Such a calling sequence is common in linkage code used when the subroutine that the programmer wants to call, here B, is in a different module from the caller; the Binder inserts “glue” code to mediate the branch.) The three branches should be as follows.
  - A calls Glue: use a \texttt{bl} or \texttt{bcl} instruction (LK=1).
  - Glue calls B: place the address of B into the Count Register, and use a \texttt{bcctr} instruction (LK=0).
  - B returns to A: use a \texttt{bclr} instruction (LK=0) (the return address is in, or can be restored to, the Link Register).

- Function call:
  Here A calls a function, the identity of which may vary from one instance of the call to another, instead of calling a specific program B. This case should be handled using the conventions of the preceding two bullets, depending on whether the call is direct or indirect, with the following differences.
  - If the call is direct, place the address of the function into the Count Register, and use a \texttt{bcctr} instruction (LK=1) instead of a \texttt{bl} or \texttt{bcl} instruction.
  - For the \texttt{bcctr[l]} instruction that branches to the function, use BH=0b11 if appropriate.
The bits corresponding to the current “a” and “t” bits, and to the current “z” bits except in the “branch always” BO encoding, had different meanings in versions of the architecture that precede Version 2.00.

- The bit corresponding to the “t” bit was called the “y” bit. The “y” bit indicated whether to use the architected default prediction (y=0) or to use the complement of the default prediction (y=1). The default prediction was defined as follows.
  - If the instruction is $bc[l]\{a\}$ with a negative value in the displacement field, the branch is taken. (This is the only case in which the prediction corresponding to the “y” bit differs from the prediction corresponding to the “t” bit.)
  - In all other cases ($bc[l]\{a\}$ with a nonnegative value in the displacement field, $bclr[l]$, or $bcctr[l]$), the branch is not taken.

- The BO encodings that test both the Count Register and the Condition Register had a “y” bit in place of the current “z” bit. The meaning of the “y” bit was as described in the preceding item.

- The “a” bit was a “z” bit.

Because these bits have always been defined either to be ignored or to be treated as hints, a given program will produce the same result on any implementation regardless of the values of the bits. Also, because even the “y” bit is ignored, in practice, by most processors that comply with versions of the architecture that precede Version 2.00, the performance of a given program on those processors will not be affected by the values of the bits.
**Branch I-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Target Addr</th>
<th>(AA=0 LK=0)</th>
<th>(AA=0 LK=1)</th>
<th>(AA=1 LK=0)</th>
<th>(AA=1 LK=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>target_addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ba</td>
<td>target_addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bl</td>
<td>target_addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bla</td>
<td>target_addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Branch Conditional B-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>BO, BI, Target Addr</th>
<th>(AA=0 LK=0)</th>
<th>(AA=0 LK=1)</th>
<th>(AA=1 LK=0)</th>
<th>(AA=1 LK=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc</td>
<td>BO, BI, target_addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bca</td>
<td>BO, BI, target_addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bl</td>
<td>BO, BI, target_addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bla</td>
<td>BO, BI, target_addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**target_addr** specifies the branch target address.

If AA=0 then the branch target address is the sum of LI || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value LI || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

**Special Registers Altered:**

- LR (if LK=1)

**Extended Mnemonics:**

Examples of extended mnemonics for Branch Conditional:

- **blt** target
- **bne** cr2, target
- **bdz** target

**Equivalent to:**

- **bc** 12,0,target
- **bc** 4,10,target
- **bc** 16,0,target

**18 LI**

if AA then NIA ←₁₀₂₃ EXTS(LI || 0b00)
else NIA ←₁₀₂₃ CIA + EXTS(LI || 0b00)
if LK then LR ←₁₀₂₃ CIA + 4

**16 BO BI BD**

if (64-bit mode) then M ← 0
else M ← 32
if ¬BO₂ then CTR ← CTR - 1
cond_ok ← BO₂ | (CTR₂₆:₂₃ ≠ 0) ⊕ BO₂
if cond_ok then
    if AA then NIA ←₁₀₂₃ EXTS(BD || 0b00)
else NIA ←₁₀₂₃ CIA + EXTS(BD || 0b00)
if LK then LR ←₁₀₂₃ CIA + 4

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 40. **target_addr** specifies the branch target address.

If AA=0 then the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value BD || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

**Special Registers Altered:**

- CTR (if BO₂=0)
- LR (if LK=1)
**Branch Conditional to Link Register**

**XL-form**

<table>
<thead>
<tr>
<th>BO</th>
<th>BI</th>
<th>BH</th>
<th>LK</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{if (64-bit mode)} & \quad \text{then } M \leftarrow 0 \\
\text{else } & \quad M \leftarrow 32 \\
\text{if } \neg BO_2 \text{ then } & \quad \text{CTR} \leftarrow \text{CTR} - 1 \\
\text{cond OK} & \quad \text{CTR}_0:61 \| 0b00 \\
\text{if LK then } & \quad LR \leftarrow \text{CIA + 4} \\
\end{align*}
\]

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 40. The BH field is used as described in Figure 42. The branch target address is LR0:61 || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

**Special Registers Altered:**

CTR (if BO=0) 
LR (if LK=1)

**Extended Mnemonics:**

Examples of extended mnemonics for Branch Conditional to Link Register:

<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>bclr 4,6</td>
<td>bclr 4,6,0</td>
</tr>
<tr>
<td>bltlr</td>
<td>bclr 12,0,0</td>
</tr>
<tr>
<td>bnelr cr2</td>
<td>bclr 4,10,0</td>
</tr>
<tr>
<td>bdnzl</td>
<td>bclr 16,0,0</td>
</tr>
</tbody>
</table>

**Programming Note**

*bclr*, *bclrl*, *bcctr*, and *bcctrl* each serve as both a basic and an extended mnemonic. The Assembler will recognize a *bclr*, *bclrl*, *bcctr*, or *bcctrl* mnemonic with three operands as the basic form, and a *bclr*, *bclrl*, *bcctr*, or *bcctrl* mnemonic with two operands as the extended form. In the extended form the BH operand is omitted and assumed to be 0b00.

---

**Branch Conditional to Count Register**

**XL-form**

<table>
<thead>
<tr>
<th>BO</th>
<th>BI</th>
<th>BH</th>
<th>LK</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{cond OK} & \quad \text{BO}_0 \| (\text{CRBI+32} = \text{BO}_2) \\
\text{if cond OK then } & \quad \text{NA} \leftarrow \text{CTR}_0:61 \| 0b00 \\
\text{if LK then } & \quad LR \leftarrow \text{CIA + 4} \\
\end{align*}
\]

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 40. The BH field is used as described in Figure 42. The branch target address is CTR0:61 || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

If the “decrement and test CTR” option is specified (BO2=0), the instruction form is invalid.

**Special Registers Altered:**

LR (if LK=1)

**Extended Mnemonics:**

Examples of extended mnemonics for Branch Conditional to Count Register:

<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcctr 4,6</td>
<td>bcctr 4,6,0</td>
</tr>
<tr>
<td>bctctr</td>
<td>bcctr 12,0,0</td>
</tr>
<tr>
<td>bctctr cr2</td>
<td>bcctr 4,10,0</td>
</tr>
</tbody>
</table>

---

*bcclr*, *bcclrl*, *bcctcr*, and *bcctcrl* each serve as both a basic and an extended mnemonic. The Assembler will recognize a *bcclr*, *bcclrl*, *bcctcr*, or *bcctcrl* mnemonic with three operands as the basic form, and a *bcclr*, *bcclrl*, *bcctcr*, or *bcctcrl* mnemonic with two operands as the extended form. In the extended form the BH operand is omitted and assumed to be 0b00.
### Branch Conditional to Branch Target

**Address Register XL-form**

<table>
<thead>
<tr>
<th>bctar</th>
<th>BO, BI, BH</th>
<th>(LK=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bctarl</td>
<td>BO, BI, BH</td>
<td>(LK=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>19</th>
<th>BO</th>
<th>BI</th>
<th>/<em>BH</em>/</th>
<th>560</th>
<th>Lk</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>19</td>
<td>31</td>
</tr>
</tbody>
</table>

If (64-bit mode)
then M ← 1
else M ← 32
if ¬BO2 then CTR ← CTR − 1
ctr_ok ← BO2 | (CTR[M:63] ≠ 0) ⊕ BO3
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then NIA ← iea TAR0:61 || 0b00
if LK then LR ← iea CIA + 4

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 40. The BH field is used as described in Figure 42. The branch target address is TAR0:61 || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

**Special Registers Altered:**

- CTR (if BO2=0)
- LR (if LK=1)

---

**Programming Note**

In some systems, the system software will restrict usage of the `bctar` instruction to only selected programs. If an attempt is made to execute the instruction when it is not available, the system error handler will be invoked. See Book III for additional information.
2.5 Condition Register Instructions

2.5.1 Condition Register Logical Instructions

The Condition Register Logical instructions have preferred forms; see Section 1.8.1. In the preferred forms, the BT and BB fields satisfy the following rule.

- The bit specified by BT is in the same Condition Register field as the bit specified by BB.

Extended mnemonics for Condition Register logical operations

A set of extended mnemonics is provided that allow additional Condition Register logical operations, beyond those provided by the basic Condition Register Logical instructions, to be coded easily. Some of these are shown as examples with the Condition Register Logical instructions. See Appendix C for additional extended mnemonics.

### Condition Register AND XL-form

**crand** BT,BA,BB

<table>
<thead>
<tr>
<th>19</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>257</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>21</td>
<td>03</td>
</tr>
</tbody>
</table>

\[ CR_{BT+32} \leftarrow CR_{BA+32} \land CR_{BB+32} \]

The bit in the Condition Register specified by BA+32 is ANDed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\( CR_{BT+32} \)

### Condition Register OR XL-form

**cror** BT,BA,BB

<table>
<thead>
<tr>
<th>19</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>449</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>21</td>
<td>01</td>
</tr>
</tbody>
</table>

\[ CR_{BT+32} \leftarrow CR_{BA+32} \lor CR_{BB+32} \]

The bit in the Condition Register specified by BA+32 is ORed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\( CR_{BT+32} \)

### Condition Register NAND XL-form

**crnand** BT,BA,BB

\[ CR_{BT+32} \leftarrow \neg (CR_{BA+32} \land CR_{BB+32}) \]

### Condition Register XOR XL-form

**crxor** BT,BA,BB

\[ CR_{BT+32} \leftarrow CR_{BA+32} \oplus CR_{BB+32} \]

**Extended Mnemonics:**

Example of extended mnemonics for Condition Register OR:

**Extended mnemonic:** crmove \( B_x, B_y \)

**Equivalent to:** cror \( B_x, B_y, B_y \)

Example of extended mnemonics for Condition Register XOR:

**Extended mnemonic:** crclr \( B_x \)

**Equivalent to:** crxor \( B_x, B_x, B_x \)
**Condition Register NOR XL-form**

\[ \text{cmnor } \quad \text{BT,BA,BB} \]

<table>
<thead>
<tr>
<th>19</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ C_{BT+32} \leftarrow \neg(\neg C_{BA+32} \lor \neg C_{BB+32}) \]

The bit in the Condition Register specified by BA+32 is ORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[ C_{BT+32} \]

**Extended Mnemonics:**

Example of extended mnemonics for *Condition Register NOR*:

**Extended mnemonic:** \[ \text{crnot } \quad Bx,By \]

**Equivalent to:** \[ \text{crnor } \quad Bx,By,By \]

---

**Condition Register AND with Complement XL-form**

\[ \text{crandc } \quad \text{BT,BA,BB} \]

<table>
<thead>
<tr>
<th>19</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>129</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ C_{BT+32} \leftarrow C_{BA+32} \land \neg C_{BB+32} \]

The bit in the Condition Register specified by BA+32 is ANDed with the complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[ C_{BT+32} \]

---

**Condition Register Equivalent XL-form**

\[ \text{creqv } \quad \text{BT,BA,BB} \]

<table>
<thead>
<tr>
<th>19</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>289</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ C_{BT+32} \leftarrow C_{BA+32} = C_{BB+32} \]

The bit in the Condition Register specified by BA+32 is XORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[ C_{BT+32} \]

**Extended Mnemonics:**

Example of extended mnemonics for *Condition Register Equivalent*:

**Extended mnemonic:** \[ \text{crset } \quad Bx \]

**Equivalent to:** \[ \text{creqv } \quad Bx,Bx,Bx \]

---

**Condition Register OR with Complement XL-form**

\[ \text{crorc } \quad \text{BT,BA,BB} \]

<table>
<thead>
<tr>
<th>19</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>417</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ C_{BT+32} \leftarrow C_{BA+32} \lor \neg C_{BB+32} \]

The bit in the Condition Register specified by BA+32 is ORed with the complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[ C_{BT+32} \]
2.5.2 Condition Register Field Instruction

*Move Condition Register Field XL-form*

\[ \text{mcrf BF,BFA} \]

<table>
<thead>
<tr>
<th></th>
<th>19</th>
<th>BF</th>
<th>//</th>
<th>BFA</th>
<th>//</th>
<th>//</th>
<th>0</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>14</td>
<td>16</td>
<td>21</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

\[ CR_4 \times BF + 32 : 4 \times BF + 35 \]

The contents of Condition Register field BFA are copied to Condition Register field BF.

**Special Registers Altered:**
- CR field BF
2.6 System Call Instructions

These instructions provide the means by which a program can call upon the system to perform a service.

---

**System Call SC-form**

```
sc   LEV
```

These instructions call the system to perform a service. A complete description of these instructions can be found in Section 4.3.1 of Book III.

The first form of the instruction (sc) provides a single system call. The second form of the instruction (scv) provides the capability for 128 unique system calls.

The use of the LEV field is described in Book III. In the first form of the instruction the LEV values greater than 1 are reserved, and bits 0:5 of the LEV field (instruction bits 20:25) are treated as a reserved field.

When control is returned to the program that executed the System Call or System Call Vectored instruction, the contents of the registers will depend on the register conventions used by the program providing the system service.

These instructions are context synchronizing (see Book III).

**Special Registers Altered:**

Dependent on the system service

---

**Programming Note**

Since the scv instruction modifies the Count Register, programs should treat the contents of the Count Register as undefined after executing this instruction. See Section 4.3 of Book III.

---

**System Call Vectored SC-form**

```
scv  LEV
```

These instructions call the system to perform a service. A complete description of these instructions can be found in Section 4.3.1 of Book III.

The first form of the instruction (scv) provides the capability for 128 unique system calls.

The use of the LEV field is described in Book III. In the first form of the instruction the LEV values greater than 1 are reserved, and bits 0:5 of the LEV field (instruction bits 20:25) are treated as a reserved field.

When control is returned to the program that executed the System Call or System Call Vectored instruction, the contents of the registers will depend on the register conventions used by the program providing the system service.

These instructions are context synchronizing (see Book III).

**Special Registers Altered:**

Dependent on the system service

---

**Programming Note**

sc serves as both a basic and an extended mnemonic. The Assembler will recognize an sc mnemonic with one operand as the basic form, and an sc mnemonic with no operand as the extended form. In the extended form the LEV operand is omitted and assumed to be 0.

In application programs the value of the LEV operand for sc should be 0.
Chapter 3. Fixed-Point Facility

3.1 Fixed-Point Facility Overview

This chapter describes the registers and instructions that make up the Fixed-Point Facility.

3.2 Fixed-Point Facility Registers

3.2.1 General Purpose Registers

All manipulation of information is done in registers internal to the Fixed-Point Facility. The principal storage internal to the Fixed-Point Facility is a set of 32 General Purpose Registers (GPRs). See Figure 43.

<table>
<thead>
<tr>
<th>GPR 0</th>
<th>GPR 1</th>
<th>...</th>
<th>GPR 30</th>
<th>GPR 31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 43. General Purpose Registers

Each GPR is a 64-bit register.

3.2.2 Fixed-Point Exception Register

The Fixed-Point Exception Register (XER) is a 64-bit register.

<table>
<thead>
<tr>
<th>XER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Figure 44. Fixed-Point Exception Register

The bit definitions for the Fixed-Point Exception Register are shown below. Here M=0 in 64-bit mode and M=32 in 32-bit mode.

The bits are set based on the operation of an instruction considered as a whole, not on intermediate results (e.g., the Subtract From Carrying instruction, the result of which is specified as the sum of three values, sets bits in the Fixed-Point Exception Register based on the entire operation, not on an intermediate sum).

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Reserved</td>
</tr>
<tr>
<td>32</td>
<td>Summary Overflow (SO)</td>
</tr>
<tr>
<td></td>
<td>The Summary Overflow bit is set to 1 whenever an instruction (except mtspr and addex) sets the Overflow bit. Once set, the SO bit remains set until it is cleared by an mtspr instruction (specifying the XER). It is not altered by Compare instructions, by addex, or by other instructions (except mtspr to the XER) that cannot overflow. Executing an mtspr instruction to the XER, supplying the values 0 for SO and 1 for OV, causes SO to be set to 0 and OV to be set to 1.</td>
</tr>
<tr>
<td>33</td>
<td>Overflow (OV)</td>
</tr>
<tr>
<td></td>
<td>The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction. The Overflow bit can also used as an independent Carry bit by using the addex with operand ( \top \equiv 0 ) instruction and avoiding other instructions that modify the Overflow bit (e.g., any XO-form instruction with ( \top \equiv 1 )). XO-form Add, Subtract From, and Negate instructions having ( \top \equiv 1 ) set it to 1 if the carry out of bit ( \top ) is not equal to the carry out of bit ( \top +1 ), and set it to 0 otherwise. XO-form Multiply Low and Divide instructions having ( \top \equiv 1 ) set it to 1 if the result cannot be represented in 64 bits (mulld, divd, divide, divdu, divdeu) or in 32 bits (mullw, divw, divwe, divwu, divweu), and set it to 0 otherwise.</td>
</tr>
</tbody>
</table>
addex with operand \(Y=0\) sets OV to 1 if there is a carry out of bit \(M\), and sets it to 0 otherwise.

The OV bit is not altered by Compare instructions, or by other instructions (except mt spr to the XER) that cannot overflow.

34 Carry (CA)
The Carry bit is set as follows, during execution of certain instructions. Add Carrying, Subtract From Carrying, Add Extended, and Subtract From Extended types of instructions set it to 1 if there is a carry out of bit \(M\), and set it to 0 otherwise. Shift Right Algebraic instructions set it to 1 if any 1-bits have been shifted out of a negative operand, and set it to 0 otherwise. The CA bit is not altered by Compare instructions, or by other instructions (except Shift Right Algebraic, mt spr to the XER) that cannot carry.

35:43 Reserved

44 Overflow32 (OV32)
OV32 is set whenever OV is implicitly set, and is set to the same value that OV is defined to be set to in 32-bit mode.

45 Carry32 (CA32)
CA32 is set whenever CA is implicitly set, and is set to the same value that CA is defined to be set to in 32-bit mode.

46:56 Reserved
Bits 48:55 are implemented, and can be read and written by software as if the bits contained a defined field.

57:63 This field specifies the number of bytes to be transferred by a Load String Indexed or Store String Indexed instruction.

### Programming Note

Bits 48:55 of the XER correspond to bits 16:23 of the XER in the POWER Architecture. In the POWER Architecture bits 16:23 of the XER contain the comparison byte for the **ls cbx** instruction. Power ISA lacks the **ls cbx** instruction, but some application programs that run on processors that implement Power ISA may still use **ls cbx**, and privileged software may emulate the instruction. XER\(_{48:55}\) may be assigned a meaning in a future version of the architecture, when POWER compatibility for **ls cbx** is no longer needed, so these bits should not be used for purposes other than the **ls cbx** comparison byte.

### 3.2.3 VR Save Register

The VR Save Register (VRSAVE) is a 32-bit register that can be used as a software use SPR; see Section 6.3.3.
3.3 Fixed-Point Facility Instructions

3.3.1 Fixed-Point Storage Access Instructions

The Storage Access instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.10.3 on page 29.

**Programming Note**

The la extended mnemonic permits computing an effective address as a Load or Store instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at that address.

3.3.1.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (Store only), or if the program attempts to access storage that is unavailable.

3.3.2 Fixed-Point Load Instructions

The byte, halfword, word, or doubleword in storage addressed by EA is loaded into register RT.

Many of the Load instructions have an "update" form, in which register RA is updated with the effective address. For these forms, if RA≠0 and RA≠RT, the effective address is placed into register RA and the storage element (byte, halfword, word, or doubleword) addressed by EA is loaded into RT.

**Programming Note**

In some implementations, the Load Algebraic and Load with Update instructions may have greater latency than other types of Load instructions. Moreover, Load with Update instructions may take longer to execute in some implementations than the corresponding pair of a non-update Load instruction and an Add instruction.
### Load Byte and Zero D-form

**lbz**  
RT,D(RA)

<table>
<thead>
<tr>
<th>34</th>
<th>RT</th>
<th>RA</th>
<th>D</th>
<th>16</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td>34</td>
</tr>
</tbody>
</table>

### Prefixed Load Byte and Zero MLS:D-form

**plbz**  
RT,D(RA),R

**Prefix:**

<table>
<thead>
<tr>
<th>34</th>
<th>RT</th>
<th>RA</th>
<th>16</th>
<th>D</th>
<th>11</th>
<th>6</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td>34</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

**Suffix:**

<table>
<thead>
<tr>
<th>34</th>
<th>RT</th>
<th>RA</th>
<th>D</th>
<th>16</th>
<th>6</th>
<th>0</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td>34</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

- If “lbz” then
  \[ EA ← (RA|0) + \text{EXT}564(D) \]
- If “plbz” & R=0 then
  \[ EA ← (RA|0) + \text{EXT}564(d0||d1) \]
- If “plbz” & R=1 then
  \[ EA ← (\text{CIA} + \text{EXT}564(d0||d1)) \]

### Load Byte and Zero Indexed X-form

**lbzx**  
RT,RA,RB

<table>
<thead>
<tr>
<th>35</th>
<th>RT</th>
<th>RA</th>
<th>16</th>
<th>D</th>
<th>16</th>
<th>6</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td>34</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

**Equivalent to:**

- \( \text{plbz} \) Rx,value(Ry)
- \( \text{plbz} \) Rx,value(Ry),0
- \( \text{plbz} \) Rx,value

### Load Byte and Zero with Update D-form

**lbzu**  
RT,D(RA)

<table>
<thead>
<tr>
<th>35</th>
<th>RT</th>
<th>RA</th>
<th>16</th>
<th>D</th>
<th>16</th>
<th>6</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td>34</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

**Equivalent to:**

- \( \text{lbz} \) Rx,value(Ry)
- \( \text{lbz} \) Rx,value(Ry),0
- \( \text{lbz} \) Rx,value

### Load Byte and Zero with Update Indexed X-form

**lbzux**  
RT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RA</th>
<th>119</th>
<th>6</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
<td>6</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

**Equivalent to:**

- \( \text{plbz} \) Rx,value(Ry)
- \( \text{plbz} \) Rx,value(Ry),0
- \( \text{plbz} \) Rx,value

---

**Extended Mnemonics:**

Extended mnemonics for **Prefixed Load Byte and Zero**:

- \( \text{plbz} \) Rx,value
- \( \text{plbz} \) Rx,value(Ry)
Load Halfword and Zero D-form

\[
\text{lhz RT, D(RA)}
\]

Prefix: \[
\text{lhz RT, D(RA)}
\]

Suffix: \[
\text{lhz RT, D(RA)}
\]

if “lhz” then
  \[
  EA \leftarrow (RA) + \text{EXTS}(D)
  \]

if “plhz” & R=0 then
  \[
  EA \leftarrow (RA) + \text{EXTS}(d0 \mid d1)
  \]

if “plhz” & R=1 then
  \[
  EA \leftarrow \text{CIA} + \text{EXTS}(d0 \mid d1)
  \]

\[
RT \leftarrow \text{EXTZ}(\text{MEM}(EA, 2))
\]

For \text{lhz}, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value D, sign-extended to 64 bits.

For \text{plhz} with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0\mid d1, sign-extended to 64 bits.

For \text{plhz} with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0\mid d1, sign-extended to 64 bits.

The halfword in storage addressed by EA is loaded into RT[48:63]. RT[0:47] are set to 0.

For \text{plhz}, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonics for \text{Prefixed Load Halfword and Zero}:

Extended mnemonic: Equivalent to:

\[
\text{plhz Rx, value(Ry)} \quad \text{plhz Rx, value(Ry), 0}
\]

\[
\text{plhz Rx, value} \quad \text{plhz Rx, value(0), 1}
\]

Load Halfword and Zero Indexed X-form

\[
\text{lhzx RT, RA, RB}
\]

if RA = 0 then b \leftarrow 0
else b \leftarrow (RA)

EA \leftarrow b + (RB)

RT \leftarrow 48 \mid \text{MEM}(EA, 2)

Let the effective address (EA) be the sum (RA) + (RB). The halfword in storage addressed by EA is loaded into RT[48:63]. RT[0:47] are set to 0.

Special Registers Altered:
None

Load Halfword and Zero with Update D-form

\[
\text{lhzu RT, D(RA)}
\]

EA \leftarrow (RA) + \text{EXTS}(D)

RT \leftarrow 48 \mid \text{MEM}(EA, 2)

RA \leftarrow EA

Let the effective address (EA) be the sum (RA) + D. The halfword in storage addressed by EA is loaded into RT[48:63]. RT[0:47] are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Halfword and Zero with Update Indexed X-form

\[
\text{lhzux RT, RA, RB}
\]

EA \leftarrow (RA) + (RB)

RT \leftarrow 48 \mid \text{MEM}(EA, 2)

RA \leftarrow EA

Let the effective address (EA) be the sum (RA) + (RB). The halfword in storage addressed by EA is loaded into RT[48:63]. RT[0:47] are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None
Load Halfword Algebraic D-form

\[ lha \ RT, D(RA) \]

Prefix:

\[
\begin{array}{cccc}
    0 & 6 & 11 & 16 \\
    42 & RT & RA & D \\
\end{array}
\]

Suffix:

\[
\begin{array}{cccc}
    0 & 6 & 11 & 16 \\
    42 & RT & RA & d0 \\
\end{array}
\]

If the halfword in storage addressed by \( EA \) is loaded into \( RT_{48:63} \), \( RT_{0:47} \) are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for Prefixed Load Halfword Algebraic:

- \( lha\ RT, D(RA) \) or \( lha\ RT, D(RA), R \)

Prefixed Load Halfword Algebraic MLS:D-form

\[ plha \ RT, D(RA), R \]

Prefix:

\[
\begin{array}{cccc}
    0 & 6 & 11 & 16 \\
    42 & RT & RA & d0 \\
\end{array}
\]

Suffix:

\[
\begin{array}{cccc}
    0 & 6 & 11 & 16 \\
    42 & RT & RA & d1 \\
\end{array}
\]

If the halfword in storage addressed by \( EA \) is loaded into \( RT_{48:63} \), \( RT_{0:47} \) are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:

None

Load Halfword Algebraic Indexed X-form

\[ lhax \ RT, RA, RB \]

If \( RA = 0 \) then \( b \leftarrow 0 \) else \( b \leftarrow (RA) \)

\[ EA \leftarrow b + (RB) \]

\[ RT \leftarrow EXTS(MEM(EA, 2)) \]

Let the effective address \( EA \) be the sum \( (RA)0 + (RB) \). The halfword in storage addressed by \( EA \) is loaded into \( RT_{48:63} \). \( RT_{0:47} \) are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:

None

Extended mnemonic: Equivalent to:

- \( plha\ Rx, value(Ry) \) or \( plha\ Rx, value(Ry), 0 \)
- \( plha\ Rx, value \) or \( plha\ Rx, value(0), 2 \)

Load Halfword Algebraic with Update D-form

\[ lhau \ RT, D(RA) \]

\[ EA \leftarrow (RA) + EXTS(D) \]

\[ RT \leftarrow EXTS(MEM(EA, 2)) \]

\[ RA \leftarrow EA \]

Let the effective address \( EA \) be the sum \( (RA) + D \). The halfword in storage addressed by \( EA \) is loaded into \( RT_{48:63} \). \( RT_{0:47} \) are filled with a copy of bit 0 of the loaded halfword.

If \( RA = 0 \) or \( RA = RT \), the instruction form is invalid.

Special Registers Altered:

None

Load Halfword Algebraic with Update Indexed X-form

\[ lhaux \ RT, RA, RB \]

\[ EA \leftarrow (RA) + (RB) \]

\[ RT \leftarrow EXTS(MEM(EA, 2)) \]

\[ RA \leftarrow EA \]

Let the effective address \( EA \) be the sum \( (RA) + (RB) \). The halfword in storage addressed by \( EA \) is loaded into \( RT_{48:63} \). \( RT_{0:47} \) are filled with a copy of bit 0 of the loaded halfword.

If \( RA = 0 \) or \( RA = RT \), the instruction form is invalid.

Special Registers Altered:

None
Load Word and Zero D-form

lwz  RT,D(RA)

<table>
<thead>
<tr>
<th>32</th>
<th>RT</th>
<th>RA</th>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

Prefixed Load Word and Zero MLS:D-form

plwz  RT,D(RA),R

Prefix:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>0</th>
<th>R</th>
<th>d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th>32</th>
<th>RT</th>
<th>RA</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

If “lwz” then

EA ← (RA|0) + EXT564(D)

If “plwz” & R=0 then

EA ← (RA|0) + EXT564(d0 || d1)

If “plwz” & R=1 then

EA ← CIA + EXT564(d0 || d1)

RT ← 320 || MEM(EA, 4)

For lwz, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value D, sign-extended to 64 bits.

For plwz with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

For plwz with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0||d1, sign-extended to 64 bits.

The word in storage addressed by EA is loaded into RT32:63. RT0:31 are set to 0.

For plwz, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for Prefixed Load Word and Zero:

Extended mnemonic:  Equivalent to:
plwz Rx, value(Ry)  plwz Rx, value(Ry), 0
plwz Rx, value    plwz Rx, value(0), 1

Load Word and Zero Indexed X-form

lwzx  RT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RA</th>
<th>23</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

If RA = 0 then b ← 0
else

b ← (RA)

EA ← b + (RB)

RT ← 320 || MEM(EA, 4)

Let the effective address (EA) be the sum (RA|0) + (RB). The word in storage addressed by EA is loaded into RT32:63. RT0:31 are set to 0.

Special Registers Altered:

None

Load Word and Zero with Update D-form

lwzu  RT,D(RA)

<table>
<thead>
<tr>
<th>33</th>
<th>RT</th>
<th>RA</th>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

EA ← (RA) + EXT564(D)

RT ← 320 || MEM(EA, 4)

RA ← EA

Let the effective address (EA) be the sum (RA) + D. The word in storage addressed by EA is loaded into RT32:63. RT0:31 are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Word and Zero with Update Indexed X-form

lwzux  RT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RA</th>
<th>55</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

EA ← (RA) + (RB)

RT ← 320 || MEM(EA, 4)

RA ← EA

Let the effective address (EA) be the sum (RA) + (RB). The word in storage addressed by EA is loaded into RT32:63. RT0:31 are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None
3.3.2.1 64-bit Fixed-Point Load Instructions

Load Word Algebraic DS-form

\[
lwa \quad RT, DS(RA)
\]

<table>
<thead>
<tr>
<th>58</th>
<th>RT</th>
<th>RA</th>
<th>DS</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>

Prefixed Load Word Algebraic 8LS:D-form

\[
plwa \quad RT, D(RA), R
\]

Prefix:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>0</td>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th>41</th>
<th>RT</th>
<th>RA</th>
<th>d1</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

- If "lwa" then
  \[ EA \leftarrow (RA|0) + \text{EXTS}64(\text{DS}|0b00) \]
- If "plwa" & R=0 then
  \[ EA \leftarrow (RA|0) + \text{EXTS}64(d0|d1) \]
- If "plwa" & R=1 then
  \[ EA \leftarrow \text{CIA} + \text{EXTS}64(d0|d1) \]

\[ RT \leftarrow \text{EXTS(MEM(EA, 4))} \]

For \textit{lwa}, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if \( RA=0 \), and the value \( DS|0b00 \), sign-extended to 64 bits.

For \textit{plwa} with \( R=0 \), let the effective address (EA) be the sum of the contents of register RA, or the value 0 if \( RA=0 \), and the value \( d0|d1 \), sign-extended to 64 bits.

For \textit{plwa} with \( R=1 \), let the effective address (EA) be the sum of the address of the instruction and the value \( d0|d1 \), sign-extended to 64 bits.

The word in storage addressed by EA is loaded into \( RT_{32:63} \). RT\(_{0:31} \) are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:

None

Load Word Algebraic Indexed X-form

\[
lwax \quad RT, RA, RB
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RA</th>
<th>341</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

- If \( RA = 0 \) then \( b \leftarrow 0 \)
- Else \( b \leftarrow (RA|0) \)
- \( EA \leftarrow b + (RB) \)
- \( RT \leftarrow \text{EXTS(MEM(EA, 4))} \)

Let the effective address (EA) be the sum \( (RA|0) + (RB) \). The word in storage addressed by EA is loaded into \( RT_{32:63} \). RT\(_{0:31} \) are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:

None

Load Word Algebraic with Update Indexed X-form

\[
lwaux \quad RT, RA, RB
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RA</th>
<th>373</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ EA \leftarrow (RA|0) + (RB) \]
\[ RT \leftarrow \text{EXTS(MEM(EA, 4))} \]
\[ RA \leftarrow EA \]

Let the effective address (EA) be the sum \( (RA|0) + (RB) \). The word in storage addressed by EA is loaded into \( RT_{32:63} \). RT\(_{0:31} \) are filled with a copy of bit 0 of the loaded word.

EA is placed into register RA.

If \( RA=0 \) or \( RA=RT \), the instruction form is invalid.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for Prefixes Load Word Algebraic:

- \( \text{plwa} \quad Ra, value(Ry) \rightarrow \text{plwa} \quad Rx, value(Ry), 0 \)
- \( \text{plwa} \quad Ra, value \rightarrow \text{plwa} \quad Rx, value(0), 1 \)

Special Registers Altered:

None
**Load Doubleword DS-form**

```
<table>
<thead>
<tr>
<th>Id</th>
<th>RT</th>
<th>RA</th>
<th>DS</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>RT</td>
<td>RA</td>
<td>DS</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>30 31</td>
</tr>
</tbody>
</table>
```

**Prefix: Load Doubleword 8LS:D-form**

```
pld  RT, D(RA), R
```

**Prefix:**

```
if "ld" then
    EA ← (RA|0) + EXTS64(DS||0b00)
else
    b ← (RA)
    EA ← b + (RB)
    RT ← MEM(EA, b)
```

Let the effective address (EA) be the sum (RA|0) + (RB). The doubleword in storage addressed by EA is loaded into RT.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Extended mnemonics for Prefixed Load Doubleword:

```
pld Rx, value(Ry)
pld Rx, value
pld Rx, value(0), 1
```

**Load Doubleword Indexed X-form**

```
ld RX, RA, RB
```

<table>
<thead>
<tr>
<th>Idx</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>21</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>30 31</td>
</tr>
</tbody>
</table>

For **ld**, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value DS||0b00, sign-extended to 64 bits.

For **pld** with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

For **pld** with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0||d1, sign-extended to 64 bits.

The doubleword in storage addressed by EA is loaded into RT.

For **pld**, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

**Special Registers Altered:**

None

**Load Doubleword with Update DS-form**

```
ldu RX, RA
```

<table>
<thead>
<tr>
<th>Id</th>
<th>RT</th>
<th>RA</th>
<th>DS</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>RT</td>
<td>RA</td>
<td>DS</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>30 31</td>
</tr>
</tbody>
</table>

EA ← (RA) + EXTS(DS||0b00)
RT ← MEM(EA, b)
RA ← EA

Let the effective address (EA) be the sum (RA) + (DS||0b00). The doubleword in storage addressed by EA is loaded into RT.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

**Special Registers Altered:**

None

**Load Doubleword with Update Indexed X-form**

```
ldux RX, RA, RB
```

<table>
<thead>
<tr>
<th>Idx</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>53</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>53</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>30 31</td>
</tr>
</tbody>
</table>

EA ← (RA) + (RB)
RT ← MEM(EA, b)
RA ← EA

Let the effective address (EA) be the sum (RA) + (RB). The doubleword in storage addressed by EA is loaded into RT.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

**Special Registers Altered:**

None
3.3.3 Fixed-Point Store Instructions

The contents of register RS are stored into the byte, halfword, word, or doubleword in storage addressed by EA.

Many of the Store instructions have an “update” form, in which register RA is updated with the effective address. For these forms, the following rules apply.

- If RA≠0, the effective address is placed into register RA.
- If RS=RA, the contents of register RS are copied to the target storage element and then EA is placed into RA (RS).
**Store Byte D-form**

```
<table>
<thead>
<tr>
<th>38</th>
<th>RS</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>
```

**Prefix: Store Byte MLS:D-form**

```
pstb  RS,D(RA),R
```

**Prefixed: Store Byte MLS:D-form**

Prefix:

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>0</th>
<th>R</th>
<th>11</th>
<th>12</th>
<th>14</th>
<th>d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>12</td>
<td>14</td>
<td>d0</td>
<td>d1</td>
</tr>
</tbody>
</table>
```

Suffix:

For "stb" then
- EA ← (RA|0) + EXT64(D)

For "pstb" & R=0 then
- EA ← b + (RB)
- MEM(EA, 1) ← (RS)56:63

For "pstb" & R=1 then
- EA ← CIA + EXT64(d0||d1)

```
(RS)56:63 are stored into the byte in storage addressed by EA.
```

**Special Registers Altered:**

None

**Extended Mnemonics:**

Extended mnemonics for **Prefixed Store Byte**:

```
<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>stb</td>
<td>stb Rx,value(Ry)</td>
</tr>
<tr>
<td>pstb</td>
<td>pstb Rx,value(Ry), 0</td>
</tr>
<tr>
<td>pstb</td>
<td>pstb Rx,value(0), 1</td>
</tr>
</tbody>
</table>
```

**Store Byte Indexed X-form**

```
stbx  RS,RA,RB
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RA</th>
<th>215</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

If RA = 0 then b ← 0
else
- b ← (RA)
- EA ← b + (RB)
- MEM(EA, 1) ← (RS)56:63

```
Let the effective address (EA) be the sum (RA|0) + (RB) . (RS)56:63 are stored into the byte in storage addressed by EA.
```

**Special Registers Altered:**

None

**Store Byte with Update D-form**

```
stbu  RS,D(RA)
```

```
<table>
<thead>
<tr>
<th>39</th>
<th>RS</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>
```

```
EA ← (RA) + EXT64(D)
MEM(EA, 1) ← (RS)56:63
RA ← EA
```

```
Let the effective address (EA) be the sum (RA) + D . (RS)56:63 are stored into the byte in storage addressed by EA.
```

EA is placed into register RA.

If RA=0 , the instruction form is invalid.

**Special Registers Altered:**

None

**Store Byte with Update Indexed X-form**

```
stbux  RS,RA,RB
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RA</th>
<th>247</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

```
EA ← (RA) + (RB)
MEM(EA, 1) ← (RS)56:63
RA ← EA
```

```
Let the effective address (EA) be the sum (RA) + (RB) . (RS)56:63 are stored into the byte in storage addressed by EA.
```

EA is placed into register RA.

If RA=0 , the instruction form is invalid.

**Special Registers Altered:**

None
**Store Halfword D-form**

sth  RS,D(RA)

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**Prefixed Store Halfword MLS:D-form**

psth  RS,D(RA),R

Prefix:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

If "sth" then

\[ \text{EA} \leftarrow (\text{RA}|0) + \text{EXT64}(D) \]

If "psth" & R=0 then

\[ \text{EA} \leftarrow (\text{RA}|0) + \text{EXT64}(d0|d1) \]

If "psth" & R=1 then

\[ \text{EA} \leftarrow \text{CIA} + \text{EXT64}(d0|d1) \]

\[ \text{MEM}(\text{EA}, 2) \leftarrow (\text{RS})_{48:63} \]

**For sth**, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value D, sign-extended to 64 bits.

For psth with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0|d1, sign-extended to 64 bits.

For psth with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0|d1, sign-extended to 64 bits.

\[ (\text{RS})_{48:63} \] are stored into the halfword in storage addressed by EA.

**For psth**, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Extended mnemonics for **Prefixed Store Halfword**:

- sth  Rx,value(Ry)
- sth  Rx,value(D)
- sth  Rx,value(D)

**Store Halfword Indexed X-form**

sthx  RS,RA,RB

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If RA = 0 then b ← 0
else b ← (RA)

\[ \text{EA} \leftarrow b + (\text{RB}) \]

\[ \text{MEM}(\text{EA}, 2) \leftarrow (\text{RS})_{48:63} \]

Let the effective address (EA) be the sum (RA|0) + (RB). (RS)_{48:63} are stored into the halfword in storage addressed by EA.

**Special Registers Altered:**

None

**Store Halfword with Update D-form**

sthu  RS,D(RA)

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

\[ \text{EA} \leftarrow (\text{RA}) + \text{EXT5}(D) \]

\[ \text{MEM}(\text{EA}, 2) \leftarrow (\text{RS})_{48:63} \]

\[ \text{RA} \leftarrow \text{EA} \]

Let the effective address (EA) be the sum (RA) + D. (RS)_{48:63} are stored into the halfword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**

None

**Store Halfword with Update Indexed X-form**

sthus  RS,RA,RB

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ \text{EA} \leftarrow (\text{RA}) + (\text{RB}) \]

\[ \text{MEM}(\text{EA}, 2) \leftarrow (\text{RS})_{48:63} \]

\[ \text{RA} \leftarrow \text{EA} \]

Let the effective address (EA) be the sum (RA) + (RB). (RS)_{48:63} are stored into the halfword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**

None
**Store Word D-form**

stw    RS,D(RA)

<table>
<thead>
<tr>
<th>36</th>
<th>RS</th>
<th>RA</th>
<th>D</th>
<th>11</th>
</tr>
</thead>
</table>

**Prefixed Store Word MLS:D-form**

pstw    RS,D(RA),R

Prefix:

| 6 | 0 | R | 11 | 14 | d0 |

Suffix:

| 36 | RS | RA | d1 | 11 |

if "stw" then
   EA ← (RA|0) + EXTS64(D)
if "pstw" & R=0 then
   EA ← b + (RA)
   MEM(EA, 4) ← |RS|32:63
if "pstw" & R=1 then
   EA ← CIA + EXTS64(d0||d1)
MEM(EA, 4) ← |RS|32:63

For stw, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value D, sign-extended to 64 bits.

For pstw with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

For pstw with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0||d1, sign-extended to 64 bits.

(RS)32:63 are stored into the word in storage addressed by EA.

For pstw, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

**Special Registers Altered:**
None

**Extended Mnemonics:**

Extended mnemonics for Prefixes Store Word:

Extended mnemonic:     Equivalent to:

pstw Rx.value(Ry)    pstw Rx.value(Ry), 0
pstw Rx.value        pstw Rx.value(0), 1

**Store Word Indexed X-form**

stwx    RS,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RA</th>
<th>151</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else
   b ← (RA)
   EA ← b + (RB)
   MEM(EA, 4) ← |RS|32:63

Let the effective address (EA) be the sum (RA) + (RB). (RS)32:63 are stored into the word in storage addressed by EA.

**Special Registers Altered:**
None

**Store Word with Update D-form**

stwu    RS,D(RA)

| 37 | RS | RA | D | 11 |

EA ← (RA) + EXTS(D)
MEM(EA, 4) ← |RS|32:63
RA ← EA

Let the effective address (EA) be the sum (RA) + D. (RS)32:63 are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**
None

**Store Word with Update Indexed X-form**

stwux    RS,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RA</th>
<th>183</th>
</tr>
</thead>
</table>

EA ← (RA) + (RB)
MEM(EA, 4) ← |RS|32:63
RA ← EA

Let the effective address (EA) be the sum (RA) + (RB). (RS)32:63 are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**
None
3.3.3.1 64-bit Fixed-Point Store Instructions

### Store Doubleword DS-form

**std**

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

### Prefixed Store Doubleword 8LS:D-form

**pstd**

Prefix:

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

- if "std" then
  - if RA=0 then b ← 0
  - else b ← (RA)
  - EA ← b + (RA)
  - MEM(EA, 8) ← (RS)

For **std**, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value DS||0b00, sign-extended to 64 bits.

For **pstd** with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

For **pstd** with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0||d1, sign-extended to 64 bits.

**Special Registers Altered:** None

### Extended Mnemonics:

Extended mnemonics for **Prefixed Store Doubleword**:

- **std** Rx, value(Ry)
- **pstd** Rx, value(Ry), 0
- **pstd** Rx, value|0, 1

### Store Doubleword Indexed X-form

**stdx**

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, B) ← (RS)

Let the effective address (EA) be the sum (RA|0) + (RB).

(RS) is stored into the doubleword in storage addressed by EA.

**Special Registers Altered:** None

### Store Doubleword with Update DS-form

**stdu**

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

EA ← (RA) + EXTS64(DS||0b00)
MEM(EA, B) ← (RS)
RA ← EA

Let the effective address (EA) be the sum (RA) + (DS||0b00).

(RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:** None
**Store Doubleword with Update Indexed X-form**

stdux  RS,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RS</td>
<td>RA</td>
<td>RA</td>
<td>181</td>
</tr>
</tbody>
</table>

\[ EA \leftarrow (RA) + (RB) \]
\[ \text{MEM}(EA, 0) \leftarrow (RS) \]
\[ RA \leftarrow EA \]

Let the effective address \( EA \) be the sum \( (RA) + (RB) \).

\( (RS) \) is stored into the doubleword in storage addressed by \( EA \).

\( EA \) is placed into register \( RA \).

If \( RA = 0 \), the instruction form is invalid.

**Special Registers Altered:**

None
3.3.4 Fixed Point Load and Store Quadword Instructions

For \( lq \), the quadword in storage addressed by \( EA \) is loaded into an even-odd pair of GPRs as follows. In Big-Endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by \( EA \) and the odd-numbered GPR is loaded with the doubleword addressed by \( EA + 8 \). In Little-Endian mode, the even-numbered GPR is loaded with the byte-reversed doubleword from storage addressed by \( EA + 8 \) and the odd-numbered GPR is loaded with the byte-reversed doubleword addressed by \( EA \).

On the other hand, for \( plq \), the quadword in storage addressed by \( EA \) is loaded into an even-odd pair of GPRs as follows. Independent of endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by \( EA \) and the odd-numbered GPR is loaded with the doubleword addressed by \( EA + 8 \).

In the preferred form of the \textit{Load Quadword} instruction \( RA \neq RTp+1 \).

For \( stq \), the contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by \( EA \) as follows. In Big-Endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by \( EA \) and the odd-numbered GPR is stored into the doubleword addressed by \( EA + 8 \). In Little-Endian mode, the even-numbered GPR is stored byte-reversed into the doubleword in storage addressed by \( EA + 8 \) and the odd-numbered GPR is stored byte-reversed into the doubleword addressed by \( EA \).

On the other hand, for \( pstq \), the contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by \( EA \) as follows. Independent of endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by \( EA \) and the odd-numbered GPR is stored into the doubleword addressed by \( EA + 8 \).

\textbf{Programming Note}

The \( lq \) and \( stq \) instructions exist primarily to permit software to access quadwords in storage "atomically"; see Section 1.4 of Book II. Because GPRs are 64 bits long, the Fixed-Point Facility on many designs is optimized for storage accesses of at most eight bytes. On such designs, the quadword atomicity required for \( lq \) and \( stq \) makes these instructions complex to implement, with the result that the instructions may perform less well on these designs than the corresponding two \textit{Load Doubleword} or \textit{Store Doubleword} instructions.

The complexity of providing quadword atomicity may be especially great for storage that is Write Through Required or Caching Inhibited (see Section 1.6 of Book II). This is why \( lq \) and \( stq \) are permitted to cause the data storage error handler to be invoked if the specified storage location is in either of these kinds of storage (see Section 3.3.1.1).
Load Quadword DQ-form

\[ \text{lq } \text{RTp, DQ(RA)} \]

<table>
<thead>
<tr>
<th>56</th>
<th>RTp</th>
<th>RA</th>
<th>DQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

Prefix:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>R</th>
<th>0</th>
<th>d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>14</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th>56</th>
<th>RTp</th>
<th>RA</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**Prefixed Load Quadword 8LS:D-form**

\[ \text{plq } \text{RTp, D(RA), R} \]

**Prefix:**

If “lq” then
\[ \text{EA} \leftarrow (\text{RA}|0) + \text{EXTS64(DQ)||0b0000} \]

If “plq” & R=0 then
\[ \text{EA} \leftarrow (\text{RA}|0) + \text{EXTS64(d0||d1)} \]

If “plq” & R=1 then
\[ \text{EA} \leftarrow \text{CIA} + \text{EXTS64(d0||d1)} \]

If Big-Endian byte ordering then
\[ \text{RTp}||\text{RTp+1} \leftarrow \text{MEM(EA,16)} \]

If “lq” and Little-Endian byte ordering then
\[ \text{RTp}||\text{RTp+1} \leftarrow \text{MEM(EA,16)} \]

If “plq” and Little-Endian byte ordering then
\[ \text{RTp+1}||\text{RTp} \leftarrow \text{MEM(EA,16)} \]

For **lq**, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value DQ||0b0000, sign-extended to 64 bits.

For **plq** with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

For **plq** with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0||d1, sign-extended to 64 bits.

For Big-Endian byte ordering, the quadword in storage addressed by EA is loaded into RTp||RTp+1.

For Big-Endian byte ordering, the quadword in storage addressed by EA is byte-reversed and loaded into RTp||RTp+1.

For **lq** and Little-Endian byte ordering, the quadword in storage addressed by EA is byte-reversed and loaded into RTp||RTp.

For **plq** and Little-Endian byte ordering, the quadword in storage addressed by EA is byte-reversed and loaded into RTp||RTp.

If RTp is odd or RTp=RA, the instruction form is invalid. If RTp=RA, an attempt to execute this instruction will invoke the system illegal instruction error handler. (The RTp=RA case includes the case of RTp=RA=0.)

The quadword in storage addressed by EA is loaded into an even-odd pair of GPRs as follows. In Big-Endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by EA and the odd-numbered GPR is loaded with the doubleword addressed by EA+8. In Little-Endian mode, the even-numbered GPR is loaded with the byte-reversed doubleword from storage addressed by EA+8 and the odd-numbered GPR is loaded with the byte-reversed doubleword addressed by EA.

For **plq**, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

**Programming Note**

In versions of the architecture prior to v2.07, this instruction was privileged.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Extended mnemonics for **Prefixed Load Quadword**:

<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{plq } \text{Rx,value(Ry)}</td>
<td>\text{plq } \text{Rx,value(Ry),0}</td>
</tr>
<tr>
<td>\text{plq } \text{Rx,value}</td>
<td>\text{plq } \text{Rx,value(0),1}</td>
</tr>
</tbody>
</table>

**Version 3.1**

Chapter 3. Fixed-Point Facility 65
Store Quadword DS-form

\[
\text{stq} \quad \text{RSp,} \text{DS(RA)}
\]

Prefix:

\[
\begin{array}{cccc}
0 & 6 & 11 & 16 \\
62 & 30 & 31 & 2
\end{array}
\]

Suffix:

\[
\begin{array}{cccc}
60 & 1 & 11 & 16 \\
62 & 0 & 6 & 14
\end{array}
\]

Prefix: 10 0 // R // d0 31

if “stq” then
EA ← (RA∥0) + EXTS64(DS∥0b00)
if “pstq” & R=0 then
EA ← (RA∥0) + EXTS64(d0∥d1)
if “pstq” & R=1 then
EA ← CIA + EXTS64(d0∥d1)
if Big-Endian byte ordering then
MEM(EA,16) ← |RSp|||RSp+1|
if “stq” and Little-Endian byte ordering then
MEM(EA,16) ← |RSp|||RSp+1|
if “pstq” and Little-Endian byte ordering then
MEM(EA,16) ← |RSp+1|||RSp|

For stq, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value DS∥0b00, sign-extended to 64 bits.

For pstq with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0∥d1, sign-extended to 64 bits.

For pstq with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0∥d1, sign-extended to 64 bits.

For Big-Endian byte ordering, the content of register pair RSp||RSp+1 is stored into the quadword in storage addressed by EA.

For Little-Endian byte ordering, the content of register pair RSp||RSp+1 is byte-reversed and stored into the quadword in storage addressed by EA.

For pstq and Little-Endian byte ordering, the content of register pair RSp+1||RSp is byte-reversed and stored into the quadword in storage addressed by EA.

If RSp is odd, the instruction form is invalid.

The contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by EA as follows. In Big-Endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by EA and the odd-numbered GPR is stored into the doubleword addressed by EA+8. In Little-Endian mode, the even-numbered GPR is stored byte-reversed into the doubleword in storage addressed by EA+8 and the odd-numbered GPR is stored byte-reversed into the doubleword addressed by EA.

For pstq, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

Programming Note

In versions of the architecture prior to V. 2.07, this instruction was privileged.

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonics for Prefixed Store Quadword:

<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>pstq Rx,value(Ry)</td>
<td>pstq Rx,value(Ry),0</td>
</tr>
<tr>
<td>pstq Rx,value</td>
<td>pstq Rx,value(0),1</td>
</tr>
</tbody>
</table>

For stq and Little-Endian byte ordering, the content of register pair RSp+1||RSp is byte-reversed and stored into the quadword in storage addressed by EA.

Extended mnemonics for Prefixed Store Quadword:
3.3.5 Fixed-Point Load and Store with Byte Reversal Instructions

Load Halfword Byte-Reverse Indexed X-form

lhbrx RT, RA, RB

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
load_data ← MEM(EA, 2)
RT ← 480 || load_data8:15 || load_data0:7

Let the effective address (EA) be the sum (RA|0) + (RB).

Bits 0:7 of the halfword in storage addressed by EA are loaded into RT56:63.

Bits 8:15 of the halfword in storage addressed by EA are loaded into RT48:55.

RT0:47 are set to 0.

Special Registers Altered:
None

Programming Note
These instructions have the effect of loading and storing data in the opposite byte ordering from that which would be used by other Load and Store instructions.

Store Halfword Byte-Reverse Indexed X-form

sthbrx RS, RA, RB

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (RS)56:63 || (RS)48:55

Let the effective address (EA) be the sum (RA|0) + (RB).

(RS)56:63 are stored into bits 0:7 of the halfword in storage addressed by EA.

(RS)48:55 are stored into bits 8:15 of the halfword in storage addressed by EA.

Special Registers Altered:
None

Programming Note
In some implementations, the Load Byte-Reverse instructions may have greater latency than other Load instructions.
### Load Word Byte-Reverse Indexed X-form

**lwbrx**

<table>
<thead>
<tr>
<th>RT</th>
<th>RA</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

\[
\text{if } \text{RA} = 0 \text{ then } b \leftarrow 0 \\
\text{else } b \leftarrow \text{RA} \\
\text{EA} \leftarrow b + (\text{RB}) \\
\text{load\_data} \leftarrow \text{MEM}(\text{EA}, \text{4}) \\
\text{RT} \leftarrow (\text{load\_data}_{24:31}) || (\text{load\_data}_{16:23}) || (\text{load\_data}_{8:15}) || (\text{load\_data}_{0:7})
\]

Let the effective address (EA) be the sum \( \text{RA}|0 + \text{RB} \).

Bits 0:7 of the word in storage addressed by EA are loaded into RT_{56:63}.

Bits 8:15 of the word in storage addressed by EA are loaded into RT_{48:55}.

Bits 16:23 of the word in storage addressed by EA are loaded into RT_{40:47}.

Bits 24:31 of the word in storage addressed by EA are loaded into RT_{32:39}.

RT_{31} are set to 0.

**Special Registers Altered:**

None

### Store Word Byte-Reverse Indexed X-form

**stwbrx**

<table>
<thead>
<tr>
<th>RS</th>
<th>RA</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

\[
\text{if } \text{RA} = 0 \text{ then } b \leftarrow 0 \\
\text{else } b \leftarrow \text{RA} \\
\text{EA} \leftarrow b + (\text{RB}) \\
\text{MEM}(\text{EA}, \text{4}) \leftarrow (\text{RS})_{56:63} || (\text{RS})_{48:55} || (\text{RS})_{40:47} || (\text{RS})_{32:39}
\]

Let the effective address (EA) be the sum \( \text{RA}|0 + \text{RB} \).

(\text{RS})_{56:63} are stored into bits 0:7 of the word in storage addressed by EA.

(\text{RS})_{48:55} are stored into bits 8:15 of the word in storage addressed by EA.

(\text{RS})_{40:47} are stored into bits 16:23 of the word in storage addressed by EA.

(\text{RS})_{32:39} are stored into bits 24:31 of the word in storage addressed by EA.

**Special Registers Altered:**

None
3.3.5.1 64-Bit Load and Store with Byte Reversal Instructions

**Load Doubleword Byte-Reverse Indexed X-form**

\[ \text{ldbrx} \quad \text{RT,RA,RB} \]

\[
\begin{array}{cccccc}
\text{if RA} = 0 \text{ then } b & \leftarrow & 0 \\
\text{else} & & b & \leftarrow & (RA) \\
\text{EA} & \leftarrow & b + (RB) \\
\text{load\_data} & \leftarrow & \text{MEM}(EA, 8) \\
\text{RT} & \leftarrow & \text{load\_data}_{56:63} \ || \ \text{load\_data}_{48:55} \\
& & & | | \ \text{load\_data}_{40:47} \ || \ \text{load\_data}_{32:39} \\
& & & | | \ \text{load\_data}_{24:31} \ || \ \text{load\_data}_{16:23} \\
& & & | | \ \text{load\_data}_{8:15} \ || \ \text{load\_data}_{0:7} \\
\end{array}
\]

Let the effective address (EA) be the sum \((RA|0) + (RB)\).

Bits 0:7 of the doubleword in storage addressed by EA are loaded into RT56:63.

Bits 8:15 of the doubleword in storage addressed by EA are loaded into RT48:55.

Bits 16:23 of the doubleword in storage addressed by EA are loaded into RT40:47.

Bits 24:31 of the doubleword in storage addressed by EA are loaded into RT32:39.

Bits 32:39 of the doubleword in storage addressed by EA are loaded into RT24:31.

Bits 40:47 of the doubleword in storage addressed by EA are loaded into RT16:23.

Bits 48:55 of the doubleword in storage addressed by EA are loaded into RT8:15.

Bits 56:63 of the doubleword in storage addressed by EA are loaded into RT0:7.

**Special Registers Altered:**

None

---

**Store Doubleword Byte-Reverse Indexed X-form**

\[ \text{stdbrx} \quad \text{RS,RA,RB} \]

\[
\begin{array}{cccccc}
\text{if RA} = 0 \text{ then } b & \leftarrow & 0 \\
\text{else} & & b & \leftarrow & (RA) \\
\text{EA} & \leftarrow & b + (RB) \\
\text{MEM}(EA, 8) & \leftarrow & (RS)_{56:63} \ || \ (RS)_{48:55} \\
& & & | | \ (RS)_{40:47} \ || \ (RS)_{32:39} \\
& & & | | \ (RS)_{24:31} \ || \ (RS)_{16:23} \\
& & & | | \ (RS)_{8:15} \ || \ (RS)_{0:7} \\
\end{array}
\]

Let the effective address (EA) be the sum \((RA|0) + (RB)\).

(RS)56:63 are stored into bits 0:7 of the doubleword in storage addressed by EA.

(RS)48:55 are stored into bits 8:15 of the doubleword in storage addressed by EA.

(RS)40:47 are stored into bits 16:23 of the doubleword in storage addressed by EA.

(RS)32:39 are stored into bits 23:31 of the doubleword in storage addressed by EA.

(RS)24:31 are stored into bits 32:39 of the doubleword in storage addressed by EA.

(RS)16:23 are stored into bits 40:47 of the doubleword in storage addressed by EA.

(RS)8:15 are stored into bits 48:55 of the doubleword in storage addressed by EA.

(RS)0:7 are stored into bits 56:63 of the doubleword in storage addressed by EA.

**Special Registers Altered:**

None
3.3.6 Fixed-Point Load and Store Multiple Instructions

Load Multiple Word D-form

`lmw RT,D(RA)`

If `RA = 0` then `b ← 0` else `b ← |RA|`

`EA ← b + EXTS(D)`

`r ← RT`

Do while `r ≤ 31`

`GPR(r)32:63 ← MEM(EA, 4)`

`r ← r + 1`

`EA ← EA + 4`

Let `n = (32 - RT)`. Let the effective address (`EA`) be the sum `|RA|0 + D`.

`n` consecutive words starting at `EA` are loaded into the low-order 32 bits of GPRs `RT` through 31. The high-order 32 bits of these GPRs are set to zero.

If `RA` is in the range of registers to be loaded, including the case in which `RA=0`, the instruction form is invalid.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode, the system alignment error handler is invoked.

Special Registers Altered:
None

Store Multiple Word D-form

`stmw RS,D(RA)`

If `RA = 0` then `b ← 0` else `b ← |RA|`

`EA ← b + EXTS(D)`

`r ← RS`

Do while `r ≤ 31`

`MEM(EA, 4) ← GPR(r)32:63`

`r ← r + 1`

`EA ← EA + 4`

Let `n = (32 - RS)`. Let the effective address (`EA`) be the sum `|RA|0 + D`.

`n` consecutive words starting at `EA` are stored from the low-order 32 bits of GPRs `RS` through 31.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode, the system alignment error handler is invoked.

Special Registers Altered:
None
3.3.7 Fixed-Point Move Assist Instructions [Phased Out]

The *Move Assist* instructions allow movement of an arbitrary sequence of bytes from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields.

The *Move Assist* instructions have preferred forms; see Section 1.8.1, “Preferred Instruction Forms” on page 24. In the preferred forms, register usage satisfies the following rules.

- $\text{RS} = 4$ or $5$
- $\text{RT} = 4$ or $5$
- last register loaded/stored $\leq 12$

For some implementations, using GPR 4 for $\text{RS}$ and $\text{RT}$ may result in slightly faster execution than using GPR 5.
**Load String Word Immediate X-form**

**lswi**  
RT, RA, NB

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>20</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>597</td>
<td>31</td>
</tr>
</tbody>
</table>

if RA = 0 then EA ← 0  
else EA ← |RA|  
if NB = 0 then n ← 32  
else n ← NB  
r ← RT - 1  
i ← 32  
do while n > 0  
if i = 32 then  
r ← r + 1 (mod 32)  
GPR[r] ← 0  
i ← i + 8  
if i = 64 then i ← 32  
EA ← EA + 1  
n ← n - 1

Let the effective address (EA) be (RA|0). Let n = NB if NB=0, n = 32 if NB=0; n is the number of bytes to load. Let nr=CEIL(n/4); nr is the number of registers to receive data.

n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr-1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr-1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode, the system alignment error handler is invoked.

**Special Registers Altered:**  
None

---

**Load String Word Indexed X-form**

**lswx**  
RT, RA, RB

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>20</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>533</td>
<td>31</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0  
else b ← |RA|  
EA ← b + (RB)  
n ← XER57:63  
r ← RT - 1  
i ← 32  
RT ← undefined  
do while n > 0  
if i = 32 then  
r ← r + 1 (mod 32)  
GPR[r] ← 0  
i ← i + 8  
if i = 64 then i ← 32  
EA ← EA + 1  
n ← n - 1

Let the effective address (EA) be the sum |RA|0 + |RB|. Let n=XER57:63; n is the number of bytes to load. Let nr=CEIL(n/4); nr is the number of registers to receive data.

If n>0, n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr-1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr-1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If n=0, the contents of register RT are undefined.

If RA or RB is in the range of registers to be loaded, including the case in which RA=0, the instruction is treated as if the instruction form were invalid. If RT=RA or RT=RB, the instruction form is invalid.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode and n>0, the system alignment error handler is invoked.

**Special Registers Altered:**  
None
**Store String Word Immediate X-form**

stswi  RS,RA,NB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>725</th>
</tr>
</thead>
</table>

if RA = 0 then EA ← 0
else
  EA ← (RA)
endif

if NB = 0 then n ← 32
else
  n ← NB
endif

r ← RS - 1
i ← 32

do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)i:i+7
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
endo

Let the effective address (EA) be (RA|0). Let n = NB if NB ≠ 0; n = 32 if NB = 0; n is the number of bytes to store. Let nr = CEIL(n/4); nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs RS through RS+nr-1. Data are stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode, the system alignment error handler is invoked.

**Special Registers Altered:**

None

**Store String Word Indexed X-form**

stswx  RS,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>661</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else
  b ← (RA)
endif

EA ← b + (RB)

n ← XER57:63

r ← RS - 1
i ← 32

do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)i:i+7
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
endo

Let the effective address (EA) be the sum (RA|0) + (RB). Let n = XER57:63; n is the number of bytes to store. Let nr = CEIL(n/4); nr is the number of registers to supply data.

If n > 0, n consecutive bytes starting at EA are stored from GPRs RS through RS+nr-1. Data are stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

If n = 0, no bytes are stored.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode and n > 0, the system alignment error handler is invoked.

**Special Registers Altered:**

None
3.3.8 Other Fixed-Point Instructions

The remainder of the fixed-point instructions use the contents of the General Purpose Registers (GPRs) as source operands, and place results into GPRs, into the Fixed-Point Exception Register (XER), and into Condition Register fields. In addition, the Trap instructions test the contents of a GPR or XER bit, invoking the system trap handler if the result of the specified test is true.

These instructions treat the source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation.

The X-form and XO-form instructions with Rc=1, and the D-form instructions addic, andi, and andis, set the first three bits of CR Field 0 to characterize the result placed into the target register. In 64-bit mode, these bits are set by signed comparison of the result to zero. In 32-bit mode, these bits are set by signed comparison of the low-order 32 bits of the result to zero.

Unless otherwise noted and when appropriate, when CR Field 0 and the XER are set they reflect the value placed into the target register.

---

**Programming Note**

Instructions with the OE bit set or that set CA and CA32 may execute slowly or may prevent the execution of subsequent instructions until the instruction has completed.
3.3.9 Fixed-Point Arithmetic Instructions

The XO-form Arithmetic instructions with Rc=1, and the D-form Arithmetic instruction `addc`, set the first three bits of CR Field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions”.

`addic`, `addic`, `subfic`, `addc`, `subfc`, `adde`, `subfe`, `addme`, `subfme`, `addze`, and `subfze` always set CA, to reflect the carry out of bit 0 in 64-bit mode and out of bit 32 in 32-bit mode. These instructions also always set CA32 to reflect the carry out of bit 32. The XO-form Arithmetic instructions set SO, OV, and OV32 when OE=1 to reflect overflow of the result. Except for the Multiply Low and Divide instructions, the setting of SO and OV is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode, while OV32 reflects overflow of the low-order 32-bit result independent of the mode. For XO-form Multiply Low and Divide instructions, the setting of SO, OV, and OV32 is mode-independent, and reflects overflow of the 64-bit result for `mulld`, `divd`, `divde`, `divdu` and `divdeu`, and overflow of the low-order 32-bit result for `mullw`, `divw`, `divwe`, `divwu`, and `divweu`.

---

**Programming Note**

Notice that CR Field 0 may not reflect the “true” (infinitely precise) result if overflow occurs.

**Extended mnemonics for addition and subtraction**

Several extended mnemonics are provided that use the Add Immediate and Add Immediate Shifted instructions to load an immediate value or an address into a target register. Some of these are shown as examples with the two instructions.

The Power ISA supplies Subtract From instructions, which subtract the second operand from the third. A set of extended mnemonics is provided that use the more “normal” order, in which the third operand is subtracted from the second, with the third operand being either an immediate field or a register. Some of these are shown as examples with the appropriate Add and Subtract From instructions.

See Appendix C for additional extended mnemonics.
Add Immediate D-form

```
addi RT,RA,SI
```

Prefixed Add Immediate MLS:D-form

```
paddi RT,RA,SI,R
```

Prefix:  
Suffix:

```
if "addi" then
    RT ← (RA|0) + EXTS64(SI)
else
    RT ← (RA) + EXTS64(SI||si0
```

For `addi`, let the sum of the contents of register RA, or the value 0 if RA=0, and the value SI, sign-extended to 64 bits, is placed into register RT.

For `paddi` with R=0, the sum of the contents of register RA, or the value 0 if RA=0, and the value si0||si1, sign-extended to 64 bits, is placed into register RT.

For `paddi` with R=1, the sum of the address of the instruction and the value si0||si1, sign-extended to 64 bits, is placed into register RT.

For `paddi`, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for `Add Immediate`:

- **Extended mnemonic:** `
  li Rx, value
  addi Rx, 0, value`

- **Equivalent to:** `
  li Rx, value
  addi Rx, 0, value`

Examples of extended mnemonics for `Prefixed Add Immediate`:

- **Extended mnemonic:** `
  paddi Rx, Ry, value
  addi Rx, 0, value, 0`

- **Equivalent to:** `
  paddi Rx, Ry, value
  addi Rx, 0, value, 0`

Add Immediate Shifted D-form

```
addis RT,RA,SI
```

The sum `(RA|0) + (SI || 0x0000)` is placed into register RT.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for `Add Immediate Shifted`:

- **Extended mnemonic:** `
  li Rx, value
  addis Rx, 0, value`

- **Equivalent to:** `
  li Rx, value
  addis Rx, 0, value`

Add PC Immediate Shifted DX-form

```
addpcis RT,D
```

The sum of `NI A + (D || 0x0000)` is placed into register RT.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for `Add PC Immediate Shifted`:

- **Extended mnemonic:** `
  li a Rx
  addpcis Rx, 0`

- **Equivalent to:** `
  li a Rx
  addpcis Rx, 0`
**Add XO-form**

```
add  RT,RA,RB (OE=0 Rc=0)
add. RT,RA,RB (OE=0 Rc=1)
addo RT,RA,RB (OE=1 Rc=0)
addo. RT,RA,RB (OE=1 Rc=1)
```

The sum \((RA) + (RB)\) is placed into register \(RT\).

**Special Registers Altered:**
- \(CR0\) (if \(Rc=1\))
- \(SO\) \(OV\) \(OV32\) (if \(OE=1\))

**Add Immediate Carrying D-form**

```
adic RT,RA,SI
```

The sum \((RA) + EXTS(SI)\) is placed into register \(RT\).

**Special Registers Altered:**
- \(CA\) \(CA32\)

**Extended Mnemonics:**

Example of extended mnemonics for **Add Immediate Carrying:**

```
Extended mnemonic: Equivalent to:
subic \(Rx, Ry, value\) addic \(Rx, Ry, -value\)
```

**Add Immediate Carrying and Record D-form**

```
adic. RT,RA,SI
```

The sum \((RA) + EXTS(SI)\) is placed into register \(RT\).

**Special Registers Altered:**
- \(CRO\) \(CA\) \(CA32\)

**Extended Mnemonics:**

Example of extended mnemonics for **Add Immediate Carrying and Record:**

```
Extended mnemonic: Equivalent to:
subic. \(Rx, Ry, value\) addic. \(Rx, Ry, -value\)
```

**Subtract From XO-form**

```
subf  RT,RA,RB (OE=0 Rc=0)
subf. RT,RA,RB (OE=0 Rc=1)
subfo RT,RA,RB (OE=1 Rc=0)
subfo. RT,RA,RB (OE=1 Rc=1)
```

The sum \(\neg(RA) + (RB) + 1\) is placed into register \(RT\).

**Special Registers Altered:**
- \(CR0\) (if \(Rc=1\))
- \(SO\) \(OV\) \(OV32\) (if \(OE=1\))

**Extended Mnemonics:**

Example of extended mnemonics for **Subtract From:**

```
Extended mnemonic: Equivalent to:
subic \(Rx, Ry, value\) addic \(Rx, Ry, -value\)
```

**Subtract From Immediate Carrying D-form**

```
subfic RT,RA,SI
```

The sum \(\neg(RA) + SI + 1\) is placed into register \(RT\).

**Special Registers Altered:**
- \(CA\) \(CA32\)

**Extended Mnemonics:**

Example of extended mnemonics for **Subtract From Immediate Carrying:**

```
Extended mnemonic: Equivalent to:
subic. \(Rx, Ry, value\) addic. \(Rx, Ry, -value\)
```
Add Carrying XO-form

\[
\begin{align*}
\text{addc} & \quad \text{RT,RA,RB} & (OE=0 \ Rc=0) \\
\text{addc} & \quad \text{RT,RA,RB} & (OE=0 \ Rc=1) \\
\text{addco} & \quad \text{RT,RA,RB} & (OE=1 \ Rc=0) \\
\text{addco} & \quad \text{RT,RA,RB} & (OE=1 \ Rc=1) \\
\end{align*}
\]

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
\hline
0 & 6 & 11 & 16 & 21 & 22 & 31 & \\
\end{array}
\]

\[
RT \leftarrow (RA) + (RB)
\]

The sum \((RA) + (RB)\) is placed into register \(RT\).

Special Registers Altered:

- CA \ CA32
- CR0 \ (if \(Rc=1\))
- SO  OV  OV32 \ (if \(OE=1\))

Subtract From Carrying XO-form

\[
\begin{align*}
\text{subfc} & \quad \text{RT,RA,RB} & (OE=0 \ Rc=0) \\
\text{subfc} & \quad \text{RT,RA,RB} & (OE=0 \ Rc=1) \\
\text{subfco} & \quad \text{RT,RA,RB} & (OE=1 \ Rc=0) \\
\text{subfco} & \quad \text{RT,RA,RB} & (OE=1 \ Rc=1) \\
\end{align*}
\]

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
\hline
0 & 6 & 11 & 16 & 21 & 22 & 31 & \\
\end{array}
\]

\[
RT \leftarrow -(RA) + (RB) + 1
\]

The sum \(-(RA) + (RB) + 1\) is placed into register \(RT\).

Special Registers Altered:

- CA \ CA32
- CR0 \ (if \(Rc=1\))
- SO  OV  OV32 \ (if \(OE=1\))

Extended Mnemonics:

Example of extended mnemonics for Subtract From Carrying:

\[
\begin{align*}
\text{Extended mnemonic:} & \quad \text{Equivalent to:} \\
\text{subc Rx,Ry,Rz} & \quad \text{subfc Rx,Rz,Ry}
\end{align*}
\]

Add Extended XO-form

\[
\begin{align*}
\text{adde} & \quad \text{RT,RA,RB} & (OE=0 \ Rc=0) \\
\text{addee} & \quad \text{RT,RA,RB} & (OE=0 \ Rc=1) \\
\text{addeo} & \quad \text{RT,RA,RB} & (OE=1 \ Rc=0) \\
\text{addeeo} & \quad \text{RT,RA,RB} & (OE=1 \ Rc=1) \\
\end{align*}
\]

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
\hline
0 & 6 & 11 & 16 & 21 & 22 & 31 & \\
\end{array}
\]

\[
RT \leftarrow (RA) + (RB) + CA
\]

The sum \((RA) + (RB) + CA\) is placed into register \(RT\).

Special Registers Altered:

- CA \ CA32
- CR0 \ (if \(Rc=1\))
- SO  OV  OV32 \ (if \(OE=1\))

Subtract From Extended XO-form

\[
\begin{align*}
\text{subfe} & \quad \text{RT,RA,RB} & (OE=0 \ Rc=0) \\
\text{subfe} & \quad \text{RT,RA,RB} & (OE=0 \ Rc=1) \\
\text{subfeo} & \quad \text{RT,RA,RB} & (OE=1 \ Rc=0) \\
\text{subfeo} & \quad \text{RT,RA,RB} & (OE=1 \ Rc=1) \\
\end{align*}
\]

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
\hline
0 & 6 & 11 & 16 & 21 & 22 & 31 & \\
\end{array}
\]

\[
RT \leftarrow -(RA) + (RB) + CA
\]

The sum \(-(RA) + (RB) + CA\) is placed into register \(RT\).

Special Registers Altered:

- CA \ CA32
- CR0 \ (if \(Rc=1\))
- SO  OV  OV32 \ (if \(OE=1\))
Add to Minus One Extended XO-form

addme  RT,RA  \[OE=0 \; Rc=0\]
addme. RT,RA  \[OE=0 \; Rc=1\]
addmeo RT,RA  \[OE=1 \; Rc=0\]
addmeo. RT,RA  \[OE=1 \; Rc=1\]

\[
\begin{array}{cccccc}
31 & 6 & 11 & 16 & OE & 212 & RC \\
0 & & & & & & 31 \\
\end{array}
\]

\[
RT \leftarrow (RA) + CA - 1
\]
The sum \((RA) + CA + 641\) is placed into register RT.

Special Registers Altered:
- CA
- CA32
- CR0
- \(if \; Rc=1\)
- SO
- OV
- OV32
- \(if \; OE=1\)

Subtract From Minus One Extended XO-form

subfme  RT,RA  \[OE=0 \; Rc=0\]
subfme. RT,RA  \[OE=0 \; Rc=1\]
subfmeo RT,RA  \[OE=1 \; Rc=0\]
subfmeo. RT,RA  \[OE=1 \; Rc=1\]

\[
\begin{array}{cccccc}
31 & 6 & 11 & 16 & OE & 212 & RC \\
0 & & & & & & 31 \\
\end{array}
\]

\[
RT \leftarrow \neg(\neg RA) + CA - 1
\]
The sum \(\neg(\neg RA) + CA + 641\) is placed into register RT.

Special Registers Altered:
- CA
- CA32
- CR0
- \(if \; Rc=1\)
- SO
- OV
- OV32
- \(if \; OE=1\)

Add to Zero Extended XO-form

addze  RT,RA  \[OE=0 \; Rc=0\]
addze. RT,RA  \[OE=0 \; Rc=1\]
addzeo RT,RA  \[OE=1 \; Rc=0\]
addzeo. RT,RA  \[OE=1 \; Rc=1\]

\[
\begin{array}{cccccc}
31 & 6 & 11 & 16 & OE & 212 & RC \\
0 & & & & & & 31 \\
\end{array}
\]

\[
RT \leftarrow (RA) + CA
\]
The sum \((RA) + CA\) is placed into register RT.

Special Registers Altered:
- CA
- CA32
- CR0
- \(if \; Rc=1\)
- SO
- OV
- OV32
- \(if \; OE=1\)

Subtract From Zero Extended XO-form

subfze  RT,RA  \[OE=0 \; Rc=0\]
subfze. RT,RA  \[OE=0 \; Rc=1\]
subfzeo RT,RA  \[OE=1 \; Rc=0\]
subfzeo. RT,RA  \[OE=1 \; Rc=1\]

\[
\begin{array}{cccccc}
31 & 6 & 11 & 16 & OE & 212 & RC \\
0 & & & & & & 31 \\
\end{array}
\]

\[
RT \leftarrow \neg(\neg RA) + CA
\]
The sum \(\neg(\neg RA) + CA\) is placed into register RT.

Special Registers Altered:
- CA
- CA32
- CR0
- \(if \; Rc=1\)
- SO
- OV
- OV32
- \(if \; OE=1\)

Programming Note

The setting of CA and CA32 by the Add and Subtract From instructions, including the Extended versions thereof, is mode-dependent. If a sequence of these instructions is used to perform extended-precision addition or subtraction, the same mode should be used throughout the sequence.
Add Extended using alternate carry bit
Z23-form

```
addex    RT,RA,RB,CY
```

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>23</th>
<th>170</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>CY</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If CY=0 then RT ← |RA| + |RB| + OV

For CY=0, the sum |RA| + |RB| + OV is placed into register RT.

For CY=0, OV is set to 1 if there is a carry out of bit 0 of the sum in 64-bit mode or there is a carry out of bit 32 of the sum in 32-bit mode, and set to 0 otherwise. OV32 is set to 1 if there is a carry out of bit 32 bit of the sum.

CY=1, CY=2, and CY=3 are reserved.

**Special Registers Altered:**

- OV
- OV32 (if CY=0)

---

**Programming Note**

An addc-equivalent instruction using OV is not provided. An equivalent capability can be emulated by first initializing OV to 0, then using addex. OV can be initialized to 0 using subfo, subtracting any operand from itself.

---

Negate XO-form

```
neg     RT,RA
neg.    RT,RA
nego    RT,RA
nego.   RT,RA
```

```

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>104</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RT</td>
<td>RA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RT ← ¬(RA) + 1 is placed into register RT.

The sum ¬(RA) + 1 is placed into register RT.

If the processor is in 64-bit mode and register RA contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative number and, if OE=1, OV is set to 1. If (RA)32:63 contain the most negative 32-bit number (0x8000_0000) and OE=1, OV32 is set to 1.

Similarly, if the processor is in 32-bit mode and (RA)32:63 contain the most negative 32-bit number (0x8000_0000), the low-order 32 bits of the result contain the most negative 32-bit number and, if OE=1, OV and OV32 are set to 1.

**Special Registers Altered:**

- CR0 (if Rc=1)
- SO OV OV32 (if OE=1)
### Multiply Low Immediate D-form

**mulli**

\[
\text{RT, RA, SI}
\]

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

\[\text{prod}_{0:127} \leftarrow (\text{RA}) \times \text{EXTS(SI)}\]

\[\text{RT} \leftarrow \text{prod}_{64:127}\]

The 64-bit first operand is \(\text{RA}\). The 64-bit second operand is the sign-extended value of the \(\text{SI}\) field. The low-order 64 bits of the 128-bit product of the operands are placed into register \(\text{RT}\).

Both operands and the product are interpreted as signed integers.

**Special Registers Altered:** None

### Multiply Low Word XO-form

**mullw**

\[\text{RT, RA, RB}\]

\[\text{(OE}=0 \text{ } \text{RC}=0)\]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>212</th>
<th>235</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>212</td>
<td>235</td>
<td>31</td>
</tr>
</tbody>
</table>

\[\text{RT} \leftarrow (\text{RA})_{32:63} \times (\text{RB})_{32:63}\]

The 32-bit operands are the low-order 32 bits of \(\text{RA}\) and \(\text{RB}\). The high-order 32 bits of the 64-bit product of the operands are placed into \(\text{RT}\).

If \(\text{OE}=1\) then \(\text{OV}\) and \(\text{OV}_{32}\) are set to 1 if the product cannot be represented in 32 bits.

Both operands and the product are interpreted as signed integers.

**Special Registers Altered:**
- CR0 (if \(\text{RC}=1\))
- SO \(\text{OV}\) \(\text{OV}_{32}\) (if \(\text{OE}=1\))

#### Programming Note

For **mulli** and **mullw**, the low-order 32 bits of the product are the correct 32-bit product for 32-bit mode.

For **mulli** and **mulld**, the low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers.

For **mulli** and **mullw**, the low-order 32 bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

### Multiply High Word XO-form

**mulhw**

\[\text{RT, RA, RB}\]

\[\text{(RC}=0)\]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>212</th>
<th>75</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>212</td>
<td>75</td>
<td>31</td>
</tr>
</tbody>
</table>

\[\text{prod}_{0:63} \leftarrow (\text{RA})_{32:63} \times (\text{RB})_{32:63}\]

\[\text{RT}_{32:63} \leftarrow \text{prod}_{0:31}\]

\[\text{RT}_{0:31} \leftarrow \text{undefined}\]

The 32-bit operands are the low-order 32 bits of \(\text{RA}\) and \(\text{RB}\). The high-order 32 bits of the 64-bit product of the operands are placed into \(\text{RT}_{32:63}\). The contents of \(\text{RT}_{0:31}\) are undefined.

Both operands and the product are interpreted as signed integers.

**Special Registers Altered:**
- CR0 (bits 0:2 undefined in 64-bit mode) (if \(\text{RC}=1\))

### Multiply High Word Unsigned XO-form

**mulhwu**

\[\text{RT, RA, RB}\]

\[\text{(RC}=0)\]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>212</th>
<th>11</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>212</td>
<td>11</td>
<td>31</td>
</tr>
</tbody>
</table>

\[\text{prod}_{0:63} \leftarrow (\text{RA})_{32:63} \times (\text{RB})_{32:63}\]

\[\text{RT}_{32:63} \leftarrow \text{prod}_{0:31}\]

\[\text{RT}_{0:31} \leftarrow \text{undefined}\]

The 32-bit operands are the low-order 32 bits of \(\text{RA}\) and \(\text{RB}\). The high-order 32 bits of the 64-bit product of the operands are placed into \(\text{RT}_{32:63}\). The contents of \(\text{RT}_{0:31}\) are undefined.

Both operands and the product are interpreted as unsigned integers, except that if \(\text{RC}=1\) the first three bits of CR Field 0 are set by signed comparison of the result to zero.

**Special Registers Altered:**
- CR0 (bits 0:2 undefined in 64-bit mode) (if \(\text{RC}=1\))
Divide Word XO-form

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>OE</th>
<th>RC</th>
<th>491</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>22</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{dividend}_{0:31} \leftarrow (RA)_{32:63}
\]
\[
\text{divisor}_{0:31} \leftarrow (RB)_{32:63}
\]
\[
RT_{32:63} \leftarrow \text{dividend} \div \text{divisor}
\]
\[
RT_{0:31} \leftarrow \text{undefined}
\]

The 32-bit dividend is \((RA)_{32:63}\). The 32-bit divisor is \((RB)_{32:63}\). The 32-bit quotient is placed into \(RT_{32:63}\). The contents of \(RT_{0:31}\) are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < \lvert \text{divisor} \rvert\) if the dividend is nonnegative, and \(-\lvert \text{divisor} \rvert < r \leq 0\) if the dividend is negative.

If an attempt is made to perform any of the divisions

\[
0 \times 0000_0000 + -1
\]
\[
<\text{anything}> + 0
\]

then the contents of register \(RT\) are undefined as are (if \(RC=1\)) the contents of the \(LT\), \(GT\), and \(EQ\) bits of CR Field 0. In these cases, if \(OE=1\) then OV and OV32 are set to 1.

Special Registers Altered:

- CR0 (bits 0:2 undefined in 64-bit mode) (if \(RC=1\))
- SO OV OV32 (if \(OE=1\))

Programming Note

The 32-bit signed remainder of dividing \((RA)_{32:63}\) by \((RB)_{32:63}\) can be computed as follows, except in the case that \((RA)_{32:63} = -2^{31}\) and \((RB)_{32:63} = -1\).

\[
\text{divw} \quad RT, RA, RB \quad \# \quad RT = \text{quotient}
\]
\[
\text{mullw} \quad RT, RT, RB \quad \# \quad RT = \text{quotient} \times \text{divisor}
\]
\[
\text{subf} \quad RT, RT, RA \quad \# \quad RT = \text{remainder}
\]
**Divide Word Extended XO-form**

\[
\text{divwe} \quad \text{RT,RA,RB} \quad (OE=0 \quad Rc=0)
\]
\[
\text{divwe} \quad \text{RT,RA,RB} \quad (OE=0 \quad Rc=1)
\]
\[
\text{divweo} \quad \text{RT,RA,RB} \quad (OE=1 \quad Rc=0)
\]
\[
\text{divweo} \quad \text{RT,RA,RB} \quad (OE=1 \quad Rc=1)
\]

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \ \\
\text{RT} & \text{RA} & \text{RB} & \text{OE} & \text{Rc} & \text{LT} & \text{GT} & \text{EQ}
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{dividend} & \text{divisor} & \text{RT} & \text{RA} & \text{RB} & \text{OE} & \text{Rc}
\end{array}
\]

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < |\text{divisor}|\) if the dividend is nonnegative, and \(|\text{divisor}| < r \leq 0\) if the dividend is negative.

If the quotient cannot be represented in 32 bits, or if an attempt is made to perform the division

\[
<\text{anything}> + 0
\]

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EO bits of CR Field 0. In these cases, if OE=1 then OV and OV32 are set to 1.

**Special Registers Altered:**

- CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
- SO
- OV
- OV32 (if OE=1)

**Divide Word Extended Unsigned XO-form**

\[
\text{divweu} \quad \text{RT,RA,RB} \quad (OE=0 \quad Rc=0)
\]
\[
\text{divweu} \quad \text{RT,RA,RB} \quad (OE=0 \quad Rc=1)
\]
\[
\text{divweuo} \quad \text{RT,RA,RB} \quad (OE=1 \quad Rc=0)
\]
\[
\text{divweuo} \quad \text{RT,RA,RB} \quad (OE=1 \quad Rc=1)
\]

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \ \\
\text{RT} & \text{RA} & \text{RB} & \text{OE} & \text{Rc}
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{dividend} & \text{divisor} & \text{RT} & \text{RA} & \text{RB} & \text{OE} & \text{Rc}
\end{array}
\]

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < |\text{divisor}|\).

If \(|\text{RA}| \geq |\text{RB}|\), or if an attempt is made to perform the division

\[
<\text{anything}> + 0
\]

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EO bits of CR Field 0. In these cases, if OE=1 then OV and OV32 are set to 1.

**Special Registers Altered:**

- CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
- SO
- OV
- OV32 (if OE=1)
Unsigned long division of a 64-bit dividend contained in two 32-bit registers by a 32-bit divisor can be computed as follows. The algorithm is shown first, followed by Assembler code that implements the algorithm. The dividend is \( Dh || Dl \), the divisor is \( Dv \), and the quotient and remainder are \( Q \) and \( R \) respectively, where these variables and all intermediate variables represent unsigned 32-bit integers. It is assumed that \( Dv > Dh \), and that assigning a value to an intermediate variable assigns the low-order 32 bits of the value and ignores any higher-order bits of the value. (In both the algorithm and the Assembler code, "r1" and "r2" refer to "remainder 1" and "remainder 2", rather than to GPRs 1 and 2.)

Algorithm:

3. \( q1 \leftarrow \text{divweu } Dh, Dv \) # remainder of step 1
divide operation (see Note 1)
4. \( r1 \leftarrow -(q1 \times Dv) \) # remainder of step 1
divide operation
5. \( q2 \leftarrow \text{divwu } Dl, Dv \) # remainder of step 2
divide operation
6. \( r2 \leftarrow Dl - (q2 \times Dv) \) # remainder of step 2
7. \( Q \leftarrow q1 + q2 \)
8. \( R \leftarrow r1 + r2 \)
9. if \( (R < r2) \) \( (R \geq Dv) \) then # (see Note 2)

   Q \leftarrow Q + 1 # increment quotient
   R \leftarrow R - Dv # decrement rem'der

Assembler Code:

```assembly
# Dh in r4, Dl in r5
# Dv in r6
divweu r3,r4,r6  # q1
divwu r7,r5,r6  # q2
mullw r8,r3,r6  # r1 = q1 * Dv
mullw r0,r7,r6  # r2 = q2 * Dv
subf r10,r0,r5  # r2 = Dl - (q2 * Dv)
add r3,r3,r7    # Q = q1 + q2
subf r4,r8,r10  # R = r1 + r2
cmplw r4,r10    # R < r2 ?
b1t *+12        # must adjust Q and R if yes
cmplw r4,r6     # R \geq Dv ?
b1t *+12        # must adjust Q and R if yes
addi r3,r3,1    # Q = Q + 1
subf r4,r6,r4    # R = R - Dv
# Quotient in r3
# Remainder in r4
```

Notes:

1. The remainder is \( Dh || 32^{32} \times (q1 \times Dv) \). Because the remainder must be less than \( Dv \) and \( Dv < 2^{32} \), the remainder is representable in 32 bits. Because the low-order 32 bits of \( Dh || 32^{32} \times q1 \) are 0s, the remainder is therefore equal to the low-order 32 bits of \( (q1 \times Dv) \). Thus assigning \( (q1 \times Dv) \) to \( r1 \) yields the correct remainder.

2. \( R \) is less than \( r2 \) (and also less than \( r1 \)) if and only if the addition at step 6 carried out of 32 bits — i.e., if and only if the correct sum could not be represented in 32 bits — in which case the correct sum is necessarily greater than \( Dv \).

Modulo Signed Word X-form

modsw RT,RA,RB

31  RT  RA  RB  779
 /|
3  4  11  16  21

\[ \text{dividend}_{31} \leftarrow (RA)_{32:63} \]
\[ \text{divisor}_{31} \leftarrow (RB)_{32:63} \]
\[ RT_{32:63} \leftarrow \text{dividend} \% \text{divisor} \]
\[ RT_{0:31} \leftarrow \text{undefined} \]

The 32-bit dividend is \((RA)_{32:63}\). The 32-bit divisor is \((RB)_{32:63}\). The 32-bit remainder of the dividend divided by the divisor is placed into \(RT_{32:63}\). The contents of \(RT_{0:31}\) are undefined. The quotient is not supplied as a result.

Both operands and the remainder are interpreted as signed integers. The remainder is the unique signed integer that satisfies

\[ \text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor}) \]

where \(0 \leq \text{remainder} < |\text{divisor}|\) if the dividend is nonnegative, and \(-|\text{divisor}| < \text{remainder} \leq 0\) if the dividend is negative.

If an attempt is made to perform any of the divisions

\[ 0x8000_0000 \% -1 \]
\[ <\text{anything}> \% 0 \]

then the contents of register \(RT\) are undefined.

Special Registers Altered:

None

Modulo Unsigned Word X-form

moduw RT,RA,RB

31  RT  RA  RB  267
 /|
3  4  11  16  21

\[ \text{dividend}_{31} \leftarrow (RA)_{32:63} \]
\[ \text{divisor}_{31} \leftarrow (RB)_{32:63} \]
\[ RT_{32:63} \leftarrow \text{dividend} \% \text{divisor} \]
\[ RT_{0:31} \leftarrow \text{undefined} \]

The 32-bit dividend is \((RA)_{32:63}\). The 32-bit divisor is \((RB)_{32:63}\). The 32-bit remainder of the dividend divided by the divisor is placed into \(RT_{32:63}\). The contents of \(RT_{0:31}\) are undefined. The quotient is not supplied as a result.

Both operands and the remainder are interpreted as unsigned integers. The remainder is the unique signed integer that satisfies

\[ \text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor}) \]

where \(0 \leq \text{remainder} < \text{divisor}\).

If an attempt is made to perform any of the divisions

\[ <\text{anything}> \% 0 \]

then the contents of register \(RT\) are undefined.

Special Registers Altered:

None
Deliver A Random Number X-form

darm RT,L

<table>
<thead>
<tr>
<th>L</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CRN0:31</td>
</tr>
<tr>
<td>1</td>
<td>CRN0:63</td>
</tr>
<tr>
<td>2</td>
<td>RRN0:63</td>
</tr>
<tr>
<td>3</td>
<td>reserved</td>
</tr>
</tbody>
</table>

RT ← random(L)

A random number is placed into register RT in a format selected by L as shown in the following table. The value 0xFFFFFFFF_FFFFFFFF indicates an error condition. For L=0, the random number range is 0:0xFFFFFFFF. For L=1 and L=2, the random number range is 0:0xFFFF_FFFFFFFE.

Special Registers Altered:

none

Programming Note

The random number generator provided by this instruction is NIST SP800-90B and SP800-90C compliant to the extent possible given the completeness of the standards at the time the hardware is designed. The random number generator provides a minimum of 0.5 bits of entropy per bit.

Programming Note

32-bit software running in an environment that does not preserve the high-order 32 bits of GPRs across invocations of the system error handler, signal handlers, event-based branch handlers, etc. may use the L=0 variant of darn and interpret the value 0xFFFFFFFF to indicate an error condition. The fact that the error condition includes the valid value 0x00000000_FFFFFFFF together with the true error value 0xFFFFFFFF_FFFFFFFF is not a problem.

Programming Note

When the error value is obtained, software is expected to repeat the operation. If a non-error value has not been obtained after several attempts, a software random number generation method should be used. The recommended number of attempts may be implementation specific. In the absence of other guidance, ten attempts should be adequate.
3.3.9.1 64-bit Fixed-Point Arithmetic Instructions

**Multiply Low Doubleword XO-form**

- `mulid` RT,RA,RB \( (OE=0 \quad Rc=0) \)
- `mulid.` RT,RA,RB \( (OE=0 \quad Rc=1) \)
- `mulido` RT,RA,RB \( (OE=1 \quad Rc=0) \)
- `mulido.` RT,RA,RB \( (OE=1 \quad Rc=1) \)

\[
\begin{array}{cccccc}
31 & 6 & 11 & 16 & 21 & 22 & 31 \\
\text{prod}_0:127 & \leftarrow (RA) \times (RB) \\
\text{RT} & \leftarrow \text{prod}_0:63
\end{array}
\]

The 64-bit operands are (RA) and (RB). The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

If OE=1 then OV and OV32 are set to 1 if the product cannot be represented in 64 bits.

Both operands and the product are interpreted as signed integers.

**Special Registers Altered:**

- CR0 (if \( Rc=1 \))
- SO OV OV32 (if \( OE=1 \))

**Programming Note**

The XO-form Multiply instructions may execute faster on some implementations if RB contains the operand having the smaller absolute value.

**Multiply High Doubleword XO-form**

- `mulhd` RT,RA,RB \( (Rc=0) \)
- `mulhd.` RT,RA,RB \( (Rc=1) \)

\[
\begin{array}{ccccccc}
31 & 6 & 11 & 16 & 21 & 22 & 31 \\
\text{prod}_0:127 & \leftarrow (RA) \times (RB) \\
\text{RT} & \leftarrow \text{prod}_0:63
\end{array}
\]

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as signed integers.

**Special Registers Altered:**

- CR0 (if \( Rc=1 \))

**Multiply High Doubleword Unsigned XO-form**

- `mulhdu` RT,RA,RB \( (Rc=0) \)
- `mulhdu.` RT,RA,RB \( (Rc=1) \)

\[
\begin{array}{ccccccc}
31 & 6 & 11 & 16 & 21 & 22 & 31 \\
\text{prod}_0:127 & \leftarrow (RA) \times (RB) \\
\text{RT} & \leftarrow \text{prod}_0:63
\end{array}
\]

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as unsigned integers, except that if \( Rc=1 \) the first three bits of CR Field 0 are set by signed comparison of the result to zero.

**Special Registers Altered:**

- CR0 (if \( Rc=1 \))
### Multiply-Add High Doubleword VA-form

```
maddhd RT, RA, RB, RC
```

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>RC</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{prod}_{0:127} & \leftarrow (RA) \times (RB) \\
\text{sum}_{0:127} & \leftarrow \text{prod} + \text{EXTS}(RC) \\
\text{RT} & \leftarrow \text{sum}_{0:63}
\end{align*}
\]

The 64-bit operands are \( (RA), (RB), \) and \( (RC) \). The 128-bit product of the operands \( (RA) \) and \( (RB) \) is added to \( (RC) \). The high-order 64 bits of the 128-bit sum are placed into register \( \text{RT} \).

All three operands and the result are interpreted as signed integers.

**Special Registers Altered:**

None

### Multiply-Add High Doubleword Unsigned VA-form

```
maddhdu RT, RA, RB, RC
```

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>RC</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{prod}_{0:127} & \leftarrow |RA| \times |RB| \\
\text{sum}_{0:127} & \leftarrow \text{prod} + \text{EXTZ}(RC) \\
\text{RT} & \leftarrow \text{sum}_{0:63}
\end{align*}
\]

The 64-bit operands are \( |RA|, |RB|, \) and \( |RC| \). The 128-bit product of the operands \( |RA| \) and \( |RB| \) is added to \( |RC| \). The high-order 64 bits of the 128-bit sum are placed into register \( \text{RT} \).

All three operands and the result are interpreted as unsigned integers.

**Special Registers Altered:**

None

### Multiply-Add Low Doubleword VA-form

```
maddld RT, RA, RB, RC
```

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>RC</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{prod}_{0:127} & \leftarrow (RA) \times (RB) \\
\text{sum}_{0:127} & \leftarrow \text{prod} + \text{EXTS}(RC) \\
\text{RT} & \leftarrow \text{sum}_{64:127}
\end{align*}
\]

The 64-bit operands are \( (RA), (RB), \) and \( (RC) \). The 128-bit product of the operands \( (RA) \) and \( (RB) \) is added to \( (RC) \). The low-order 64 bits of the 128-bit sum are placed into register \( \text{RT} \).

All three operands and the result are interpreted as signed integers.

**Special Registers Altered:**

None
### Divide Doubleword XO-form

**Dividend**: 
\[ (RA) \]

**Divisor**: 
\[ (RB) \]

The 64-bit dividend is \((RA)\). The 64-bit divisor is \((RB)\). The 64-bit quotient is placed into register \(RT\). The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

\[
dividend = \text{quotient} \times \text{divisor} + r
\]

where \(0 \leq r < |\text{divisor}|\) if the dividend is nonnegative, and \(-|\text{divisor}| < r \leq 0\) if the dividend is negative.

If an attempt is made to perform any of the divisions

\[
\begin{align*}
0\times8000_0000_0000_0000 & \div -1 \\
<\text{anything}> & \div 0
\end{align*}
\]

then the contents of register \(RT\) are undefined as are (if \(Rc=1\)) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if \(OE=1\) then OV and OV32 are set to 1.

**Special Registers Altered:**

- \(CR0\) (if \(Rc=1\))
- \(SO\) \(OV\) \(OV32\) (if \(OE=1\))

**Programming Note**

The 64-bit signed remainder of dividing \((RA)\) by \((RB)\) can be computed as follows, except in the case that \((RA) = -2^{63}\) and \((RB) = -1\).

\[
\begin{align*}
divd & \quad RT,RA,RB \quad \# RT = \text{quotient} \\
\text{mulld} & \quad RT,RT,RB \quad \# RT = \text{quotient} \times \text{divisor} \\
\text{subf} & \quad RT,RT,RA \quad \# RT = \text{remainder}
\end{align*}
\]
**Divide Doubleword Extended XO-form**

| divide     | RT,RA,RB | OE=0 Rc=0 |
| divide.    | RT,RA,RB | OE=0 Rc=1 |
| divideo    | RT,RA,RB | OE=1 Rc=0 |
| divideuo   | RT,RA,RB | OE=1 Rc=1 |

\[
\begin{array}{cccccc}
  31 & 30 & 29 & 28 & 27 & 26 & 25 \\
  6 & 11 & 16 & 21 & 22 & 23 & 31 \\
\end{array}
\]

\[
\text{dividend}_0\leftarrow (RA) || 640 \\
\text{divisor}_0\leftarrow (RB) \\
RT\leftarrow \text{dividend} + \text{divisor}
\]

The 128-bit dividend is \(|RA| || 640\). The 64-bit divisor is \(|RB|\). If the quotient can be represented in 64 bits, it is placed into register \(RT\). The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < |\text{divisor}|\) if the dividend is nonnegative, and \(|\text{divisor}| < r \leq 0\) if the dividend is negative.

If the quotient cannot be represented in 64 bits, or if an attempt is made to perform the division

\[
\text{<anything>} + 0
\]

then the contents of register \(RT\) are undefined as are (if \(Rc=1\)) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if \(OE=1\) then OV and OV32 are set to 1.

**Special Registers Altered:**

- CR0
- SO
- OV
- OV32

\(\text{if } Rc=1\)

\(\text{if } OE=1\)

**Divide Doubleword Extended Unsigned XO-form**

| divideu   | RT,RA,RB | OE=0 Rc=0 |
| divideu.  | RT,RA,RB | OE=0 Rc=1 |
| divideuo  | RT,RA,RB | OE=1 Rc=0 |
| divideuao | RT,RA,RB | OE=1 Rc=1 |

\[
\begin{array}{cccccc}
  31 & 30 & 29 & 28 & 27 & 26 & 25 \\
  6 & 11 & 16 & 21 & 22 & 23 & 31 \\
\end{array}
\]

\[
\text{dividend}_0\leftarrow (RA) || 640 \\
\text{divisor}_0\leftarrow (RB) \\
RT\leftarrow \text{dividend} + \text{divisor}
\]

The 128-bit dividend is \(|RA| || 640\). The 64-bit divisor is \(|RB|\). If the quotient can be represented in 64 bits, it is placed into register \(RT\). The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if \(Rc=1\) the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < \text{divisor}\).

If \(|RA| \geq |RB|\), or if an attempt is made to perform the division

\[
\text{<anything>} + 0
\]

then the contents of register \(RT\) are undefined as are (if \(Rc=1\)) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if \(OE=1\) then OV and OV32 are set to 1.

**Special Registers Altered:**

- CR0
- SO
- OV
- OV32

\(\text{if } Rc=1\)

\(\text{if } OE=1\)

**Programming Note**

Unsigned long division of a 128-bit dividend contained in two 64-bit registers by a 64-bit divisor can be accomplished using the technique described in the Programming Note with the \texttt{divweu} instruction description: \texttt{divd[e]} would be used instead of \texttt{divw[e]} (and \texttt{cmpld} instead of \texttt{cmplw}, etc.).
**Modulo Signed Doubleword X-form**

\[ \text{modsd} \quad RT, RA, RB \]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>777</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
\text{dividend} \leftarrow (RA) \\
\text{divisor} \leftarrow (RB) \\
RT \leftarrow \text{dividend} \% \text{divisor}
\]

The 64-bit dividend is \((RA)\). The 64-bit divisor is \((RB)\). The 64-bit remainder of the dividend divided by the divisor is placed into register \(RT\). The quotient is not supplied as a result.

Both operands and the remainder are interpreted as signed integers. The remainder is the unique signed integer that satisfies

\[
\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})
\]

where \(0 \leq \text{remainder} < |\text{divisor}|\) if the dividend is nonnegative, and \(-|\text{divisor}| < \text{remainder} \leq 0\) if the dividend is negative.

If an attempt is made to perform any of the divisions

\[
\text{<anything>} \% 0 \\
\text{0x8000_0000_0000_0000} \% -1
\]

then the contents of register \(RT\) are undefined.

**Special Registers Altered:**

None

---

**Modulo Unsigned Doubleword X-form**

\[ \text{modud} \quad RT, RA, RB \]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>265</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
\text{dividend} \leftarrow (RA) \\
\text{divisor} \leftarrow (RB) \\
RT \leftarrow \text{dividend} \% \text{divisor}
\]

The 64-bit dividend is \((RA)\). The 64-bit divisor is \((RB)\). The 64-bit remainder of the dividend divided by the divisor is placed into register \(RT\). The quotient is not supplied as a result.

Both operands and the remainder are interpreted as unsigned integers. The remainder is the unique signed integer that satisfies

\[
\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})
\]

where \(0 \leq \text{remainder} < \text{divisor}\).

If an attempt is made to perform any of the divisions

\[
\text{<anything>} \% 0
\]

then the contents of register \(RT\) are undefined.

**Special Registers Altered:**

None
3.3.10 Fixed-Point Compare Instructions

The fixed-point \textit{Compare} instructions compare the contents of register RA with (1) the sign-extended value of the SI field, (2) the zero-extended value of the UI field, or (3) the contents of register RB. The comparison is signed for \textit{cmpi} and \textit{cmp}, and unsigned for \textit{cmpli} and \textit{cmpl}.

The L field controls whether the operands are treated as 64-bit or 32-bit quantities, as follows:

\begin{tabular}{ll}
L & Operand length \\
0 & 32-bit operands \\
1 & 64-bit operands \\
\end{tabular}

When the operands are treated as 32-bit signed quantities, bit 32 of the register (RA or RB) is the sign bit.

The \textit{Compare} instructions set one bit in the leftmost three bits of the designated CR field to 1, and the other two to 0. XER$_{SO}$ is copied to bit 3 of the designated CR field.

The CR field is set as follows:

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LT ($</td>
</tr>
<tr>
<td>1</td>
<td>GT ($</td>
</tr>
<tr>
<td>2</td>
<td>EQ ($</td>
</tr>
<tr>
<td>3</td>
<td>SO Summary Overflow from the XER</td>
</tr>
</tbody>
</table>

Extended mnemonics for compares

A set of extended mnemonics is provided so that compares can be coded with the operand length as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the \textit{Compare} instructions. See Appendix C for additional extended mnemonics.
**Compare Immediate D-form**

\[ \text{cmpi } BF, L, RA, SI \]

<table>
<thead>
<tr>
<th>BF</th>
<th>L</th>
<th>RA</th>
<th>SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>6</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

if \( L = 0 \) then \( a \leftarrow \text{EXTS}(RA)_{32:63} \)
else \( a \leftarrow RA \)
if \( a < \text{EXTS}(SI) \) then \( c \leftarrow 0b100 \)
else if \( a > \text{EXTS}(SI) \) then \( c \leftarrow 0b010 \)
else \( c \leftarrow 0b001 \)

\[ \text{CR4} \times BF+32:4 \times BF+35 \leftarrow c || XERSO \]

The contents of register \( RA \) \((RA)_{32:63}\) sign-extended to 64 bits if \( L = 0 \) are compared with the sign-extended value of the \( SI \) field, treating the operands as signed integers. The result of the comparison is placed into CR field \( BF \).

**Special Registers Altered:**
CR field \( BF \)

**Extended Mnemonics:**

Examples of extended mnemonics for Compare Immediate:

- cmpdi Rx, value
- cmpi 0,1, Rx, value
- cmpldi Rx, value
- cmpli 0,1, Rx, value

**Compare Logical Immediate D-form**

\[ \text{cmpli } BF, L, RA, UI \]

<table>
<thead>
<tr>
<th>BF</th>
<th>L</th>
<th>RA</th>
<th>UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>6</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

if \( L = 0 \) then \( a \leftarrow 320 || (RA)_{32:63} \)
else \( a \leftarrow (RA) \)
if \( a < 480 || UI \) then \( c \leftarrow 0b100 \)
else if \( a > 480 || UI \) then \( c \leftarrow 0b010 \)
else \( c \leftarrow 0b001 \)

\[ \text{CR4} \times BF+32:4 \times BF+35 \leftarrow c || XERSO \]

The contents of register \( RA \) \((RA)_{32:63}\) zero-extended to 64 bits if \( L = 0 \) are compared with \( 480 || UI \), treating the operands as unsigned integers. The result of the comparison is placed into CR field \( BF \).

**Special Registers Altered:**
CR field \( BF \)

**Extended Mnemonics:**

Examples of extended mnemonics for Compare Logical Immediate:

- cmpldi Rx, value
- cmpli 0,1, Rx, value
- cmplw cr3, Rx, value
- cmpli 3,0, Rx, value

**Compare Logical X-form**

\[ \text{cmp } BF, L, RA, RB \]

<table>
<thead>
<tr>
<th>BF</th>
<th>L</th>
<th>RA</th>
<th>RB</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if \( L = 0 \) then \( a \leftarrow 320 || (RA)_{32:63} \)
else \( a \leftarrow (RA) \)
if \( a < (RB)_{32:63} \) then \( c \leftarrow 0b100 \)
else if \( a > (RB)_{32:63} \) then \( c \leftarrow 0b010 \)
else \( c \leftarrow 0b001 \)

\[ \text{CR4} \times BF+32:4 \times BF+35 \leftarrow c || XERSO \]

The contents of register \( RA \) \((RA)_{32:63}\) are compared with the contents of register \( RB \) \((RB)_{32:63}\) if \( L = 0 \), treating the operands as unsigned integers. The result of the comparison is placed into CR field \( BF \).

**Special Registers Altered:**
CR field \( BF \)

**Extended Mnemonics:**

Examples of extended mnemonics for Compare Logical:

- cmpld Rx, Ry
- cmpd 0,1, Rx, Ry
- cmplw cr3, Rx, Ry
- cmpli 3,0, Rx, Ry

**Chapter 3. Fixed-Point Facility**
### 3.3.10.1 Character-Type Compare Instructions

#### Compare Ranged Byte X-form

**cmprb** \(BF, L, RA, RB\)

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>BF</th>
<th>L</th>
<th>RA</th>
<th>RB</th>
<th>192</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>6</td>
<td>10</td>
<td>16</td>
<td>21</td>
<td>21</td>
</tr>
</tbody>
</table>

- **src1** ← EXTZ((RA)56:63)
- **src21hi** ← EXTZ((RB)32:39)
- **src21lo** ← EXTZ((RB)40:47)
- **src22hi** ← EXTZ((RB)48:55)
- **src22lo** ← EXTZ((RB)56:63)

if \(L=0\) then
  \[\text{in\_range} \leftarrow (\text{src22lo} \leq \text{src1}) \land (\text{src1} \leq \text{src22hi})\]
else
  \[\text{in\_range} \leftarrow (\text{src21lo} \leq \text{src1}) \land (\text{src1} \leq \text{src21hi}) \lor \text{in\_range} \leftarrow (\text{src22lo} \leq \text{src1}) \land (\text{src1} \leq \text{src22hi})\]

\[\text{CR}_{4\times BF+32:4\times BF+35} \leftarrow 0b0 || \text{in\_range} || 0b00\]

Let **src1** be the unsigned integer value in bits 56:63 of register RA.

Let **src21hi** be the unsigned integer value in bits 32:39 of register RB.

Let **src21lo** be the unsigned integer value in bits 40:47 of register RB.

Let **src22hi** be the unsigned integer value in bits 48:55 of register RB.

Let **src22lo** be the unsigned integer value in bits 56:63 of register RB.

Let \(x\) be considered “in range” of \(y:z\) if the value \(x\) is greater than or equal to the value \(y\) and the value \(x\) is less than or equal to the value \(z\).

When \(L=0\), the value \(\text{in\_range}\) is set to 1 if **src1** is in range of **src22lo**:**src22hi**. Otherwise, the value \(\text{in\_range}\) is set to 0.

When \(L=1\), the value \(\text{in\_range}\) is set to 1 if either **src1** is in range of **src21lo**:**src21hi**, or **src1** is in range of **src22lo**:**src22hi**. Otherwise, the value \(\text{in\_range}\) is set to 0.

**CR** field \(BF\) is set to the value \(0b0\) concatenated with \(\text{in\_range}\) concatenated with \(0b00\).

**Special Registers Altered:**

**CR** field \(BF\)

---

**Programming Note**

**cmprb** is useful for implementing character typing functions such as \(\text{isalpha}()\), \(\text{isdigit}()\), \(\text{isupper}()\), and \(\text{islower}()\) that are implemented using one or two range compares of the character.

A single-range compare can be implemented with an \(\text{addi}\) to load the upper and lower bounds in the range, such as \(\text{isdigit}()\).

```asm
addi    rRNG,0,0x3930    ; loads ASCII values for '9' 
&;  and '0' into rRNG
cmprb   crTGT,0,rCHAR,rRNG  ; perform range compare 
&;   sets CR field TGT to  ; indicate in range
```

A combination of \(\text{addi-\text{addis}}\) can be used to set up 2 ranges, such as for \(\text{isalpha}()\).

```asm
addi    rRNG,0,0x7A61    ; loads ASCII values for 'z' 
&;   and 'a' into rRNG
addis   rRNG,rRNG,0x5A41    ; appends ASCII values for 'Z' 
&;   and 'A' into rRNG
cmprb   crTGT,1,rCHAR,rRNG  ; perform range compare on 
&;   character in rCHAR,  ; indicate in range
```

---

**Let** **src1** be the unsigned integer value in bits 56:63 of register RA.

**Let** **src21hi** be the unsigned integer value in bits 32:39 of register RB.

**Let** **src21lo** be the unsigned integer value in bits 40:47 of register RB.

**Let** **src22hi** be the unsigned integer value in bits 48:55 of register RB.

**Let** **src22lo** be the unsigned integer value in bits 56:63 of register RB.

**Let** \(x\) be considered “in range” of \(y:z\) if the value \(x\) is greater than or equal to the value \(y\) and the value \(x\) is less than or equal to the value \(z\).

When \(L=0\), the value **in\_range** is set to 1 if **src1** is in range of **src22lo**:**src22hi**. Otherwise, the value **in\_range** is set to 0.

When \(L=1\), the value **in\_range** is set to 1 if either **src1** is in range of **src21lo**:**src21hi**, or **src1** is in range of **src22lo**:**src22hi**. Otherwise, the value **in\_range** is set to 0.

**CR** field **BF** is set to the value **0b0** concatenated with **in\_range** concatenated with **0b00**.
**Compare Equal Byte X-form**

`cmpeqb BF,RA,RB`

<table>
<thead>
<tr>
<th>31</th>
<th>BF</th>
<th>224</th>
<th>RA</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

\( src1 \leftarrow \text{GPR}[RA].\text{bit}[56:63] \)

\( match \leftarrow (src1 = (RB)00:07) \)
\( (src1 = (RB)08:15) \)
\( (src1 = (RB)16:23) \)
\( (src1 = (RB)24:31) \)
\( (src1 = (RB)32:39) \)
\( (src1 = (RB)40:47) \)
\( (src1 = (RB)48:55) \)
\( (src1 = (RB)56:63) \)

\( CR4\times BF+32:4\times BF+35 \leftarrow 0b0 || match || 0b10 \)

The `BF` field is set to indicate if the contents of bits 56:63 of register `RA` are equal to the contents of any of the 8 bytes in register `RB`. Results are undefined in 32-bit mode.

**Special Registers Altered:**

- CR field BF

---

**Programming Note**

`cmpeqb` is useful for implementing character typing functions such as `isspace()` that are implemented by comparing the character to 1 or more values.

A function such as `isspace()` can be implemented by loading the 6 byte codes corresponding to characters considered as whitespace (HT, LF, VT, FF, CR, and SP) and using the `cmpeb` to compare the subject character to those 6 values to determine if any match occurs.

```assembly
ldx rSPC, WS_CHARS ; rSPC = 0x0909_090A_0B0C_0D20
            ; load rSPC with all 6 ASCII values corresponding to
            ; white spaces
            ; perform match compare on
            ; character in rCHAR with
            ; byte values in rSPC
```

In this case, the byte code for HT (0x09) was replicated to fill the all 8 bytes to avoid a potential miscompare.
3.3.11 Fixed-Point Trap Instructions

The Trap instructions are provided to test for a specified set of conditions. If any of the conditions tested by a Trap instruction are met, the system trap handler is invoked. If none of the tested conditions are met, instruction execution continues normally.

The contents of register RA are compared with either the sign-extended value of the SI field or the contents of register RB, depending on the Trap instruction. For *tdi* and *td*, the entire contents of RA (and RB) participate in the comparison; for *twi* and *tw*, only the contents of the low-order 32 bits of RA (and RB) participate in the comparison.

This comparison results in five conditions which are ANDed with TO. If the result is not 0 the system trap handler is invoked. These conditions are as follows.

<table>
<thead>
<tr>
<th>TO Bit</th>
<th>ANDed with Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Less Than, using signed comparison</td>
</tr>
<tr>
<td>1</td>
<td>Greater Than, using signed comparison</td>
</tr>
<tr>
<td>2</td>
<td>Equal</td>
</tr>
<tr>
<td>3</td>
<td>Less Than, using unsigned comparison</td>
</tr>
<tr>
<td>4</td>
<td>Greater Than, using unsigned comparison</td>
</tr>
</tbody>
</table>

Extended mnemonics for traps

A set of extended mnemonics is provided so that traps can be coded with the condition as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the Trap instructions. See Appendix C for additional extended mnemonics.
**Trap Word Immediate D-form**

```
twi TO,RA,SI  

<table>
<thead>
<tr>
<th>3</th>
<th>TO</th>
<th>RA</th>
<th>SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>31</td>
<td>21</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>
```

*a ← EXTS((RA)32:63)*

- if (a < EXTS(SI)) & TO0 then TRAP
- if (a > EXTS(SI)) & TO1 then TRAP
- if (a = EXTS(SI)) & TO2 then TRAP
- if (a < EXT(SI)) & TO3 then TRAP
- if (a > EXT(SI)) & TO4 then TRAP

The contents of RA32:63 are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

**Special Registers Altered:**
- None

**Extended Mnemonics:**

Examples of extended mnemonics for Trap Word Immediate:

- **Extended mnemonic:** `twi Rx, value`
  - **Equivalent to:** `tw 8, Rx, value`

- **Extended mnemonic:** `twlei Rx, value`
  - **Equivalent to:** `tw 6, Rx, value`

**Trap Word X-form**

```
tw TO,RA,RB  

<table>
<thead>
<tr>
<th>31</th>
<th>TO</th>
<th>RA</th>
<th>RB</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>31</td>
<td>21</td>
<td>16</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
```

*a ← EXTS((RA)32:63)*

*b ← EXTS((RB)32:63)*

- if (a < b) & TO0 then TRAP
- if (a > b) & TO1 then TRAP
- if (a = b) & TO2 then TRAP
- if (a < u b) & TO3 then TRAP
- if (a > u b) & TO4 then TRAP

The contents of RA32:63 are compared with the contents of RB32:63. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

**Special Registers Altered:**
- None

**Extended Mnemonics:**

Examples of extended mnemonics for Trap Word:

- **Extended mnemonic:** `tweq Rx, Ry`
  - **Equivalent to:** `tw 4, Rx, Ry`

- **Extended mnemonic:** `twge Rx, Ry`
  - **Equivalent to:** `tw 5, Rx, Ry`

- **Extended mnemonic:** `trap`
  - **Equivalent to:** `tw 31, 0, 0`
### 3.3.11.1 64-bit Fixed-Point Trap Instructions

#### Trap Doubleword Immediate D-form

\[
\begin{array}{|c|c|c|c|c|}
\hline
& TO & RA & SI & \\hline
0 & 2 & 6 & 11 & 16 & 31 \\hline
\end{array}
\]

\[
\begin{align*}
a & \leftarrow (RA) \\
b & \leftarrow \text{EXTS}(SI) \\
\text{if} \ (a < b) \ & \text{&} \ \text{TO}_0 \ \text{then} \ \text{TRAP} \\
\text{if} \ (a > b) \ & \text{&} \ \text{TO}_1 \ \text{then} \ \text{TRAP} \\
\text{if} \ (a = b) \ & \text{&} \ \text{TO}_2 \ \text{then} \ \text{TRAP} \\
\text{if} \ (a <^u b) \ & \text{&} \ \text{TO}_3 \ \text{then} \ \text{TRAP} \\
\text{if} \ (a >^u b) \ & \text{&} \ \text{TO}_4 \ \text{then} \ \text{TRAP} \\
\end{align*}
\]

The contents of register RA are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

**Special Registers Altered:**
None

**Extended Mnemonics:**
Examples of extended mnemonics for Trap Doubleword Immediate:

\[
\begin{align*}
\text{tdi} & \text{lti} \ \text{Ra,value} & \text{tdi} & \text{16,Ra,value} \\
\text{tdi} & \text{nei} \ \text{Ra,value} & \text{tdi} & \text{24,Ra,value} \\
\end{align*}
\]

#### Trap Doubleword X-form

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
& TO & RA & RB & 68 & \\hline
0 & 31 & 6 & 11 & 16 & 21 & 26 & 31 \\hline
\end{array}
\]

\[
\begin{align*}
a & \leftarrow (RA) \\
b & \leftarrow (RB) \\
\text{if} \ (a < b) \ & \text{&} \ \text{TO}_0 \ \text{then} \ \text{TRAP} \\
\text{if} \ (a > b) \ & \text{&} \ \text{TO}_1 \ \text{then} \ \text{TRAP} \\
\text{if} \ (a = b) \ & \text{&} \ \text{TO}_2 \ \text{then} \ \text{TRAP} \\
\text{if} \ (a <^u b) \ & \text{&} \ \text{TO}_3 \ \text{then} \ \text{TRAP} \\
\text{if} \ (a >^u b) \ & \text{&} \ \text{TO}_4 \ \text{then} \ \text{TRAP} \\
\end{align*}
\]

The contents of register RA are compared with the contents of register RB. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

**Special Registers Altered:**
None

**Extended Mnemonics:**
Examples of extended mnemonics for Trap Doubleword:

\[
\begin{align*}
\text{Extended mnemonic:} & \quad \text{Equivalent to:} \\
\text{td} & \text{ge} \ Rx,Ry & \text{td} & \text{12,Rx,Ry} \\
\text{td} & \text{lnl} \ Rx,Ry & \text{td} & \text{5,Rx,Ry} \\
\end{align*}
\]

### 3.3.12 Fixed-Point Select

#### Integer Select A-form

\[
\begin{array}{|c|c|c|c|c|}
\hline
& RT & RA & RB & BC & 15 & \\hline
0 & 31 & 6 & 11 & 16 & 21 & 26 \\hline
\end{array}
\]

\[
\begin{align*}
\text{if} \ RA = 0 \ & \text{then} \ a \leftarrow 0 \ \text{else} \ a \leftarrow (RA) \\
\text{if} \ \text{CH} & \text{BC+32} \ & \text{then} \\
\ & \text{RT} \leftarrow a \\
\text{else} \\
\ & \text{RT} \leftarrow (RB) \\
\end{align*}
\]

If the contents of bit BC+32 of the Condition Register are equal to 1, then the contents of register RA (or 0) are placed into register RT. Otherwise, the contents of register RB are placed into register RT.

**Special Registers Altered:**
None

**Extended Mnemonics:**
Examples of extended mnemonics for Integer Select:

\[
\begin{align*}
\text{Extended mnemonic:} & \quad \text{Equivalent to:} \\
\text{isel} & \text{lt} \ Rx,Ry,Rz & \text{isel} & \text{Rx,Ry,Rz,0} \\
\text{isel} & \text{gt} \ Rx,Ry,Rz & \text{isel} & \text{Rx,Ry,Rz,1} \\
\text{isel} & \text{eq} \ Rx,Ry,Rz & \text{isel} & \text{Rx,Ry,Rz,2} \\
\end{align*}
\]
3.3.13 Fixed-Point Logical Instructions

The *Logical* instructions perform bit-parallel operations on 64-bit operands.

The X-form Logical instructions with $Rc=1$, and the D-form *Logical* instructions *andi.* and *andis.*, set the first three bits of CR Field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions” on page 74. The Logical instructions do not change the $S0$, $OV$, $OV32$, $CA$, and $CA32$ bits in the XER.

**Extended mnemonics for logical operations**

Extended mnemonics are provided that generate two different types of "no-ops" (instructions that do nothing). The first type is the preferred form, which is optimized to minimize its use of the processor's execution resources. This form is based on the *OR Immediate* instruction. The second type is the executed form, which is intended to consume the same amount of the processor's execution resources as if it were not a no-op. This form is based on the *XOR Immediate* instruction. (There are also no-ops that have other uses, such as affecting program priority, for which extended mnemonics have not been defined.)

Extended mnemonics are provided that use the *OR* and *NOR* instructions to copy the contents of one register to another, with and without complementing. These are shown as examples with the two instructions.

See Appendix C, “Assembler Extended Mnemonics” on page 1011 for additional extended mnemonics.

---

**Programming Note**

Warning: Some forms of no-op may have side effects such as affecting program priority. Programmers should use the preferred no-op unless the side effects of some other form of no-op are intended.

---

**AND Immediate D-form**

\[
\text{andi. } RA,RS,UI
\]

\[
\begin{array}{cccccc}
28 & RS & RA & UI & & 31 \\
\hline
0 & 6 & 11 & 16 & & 31 \\
\end{array}
\]

\[RA \leftarrow (RS) \& (480 || UI)\]

The contents of register \(RS\) are ANDed with \(480 || UI\) and the result is placed into register \(RA\).

**Special Registers Altered:**

\(CR0\)

**AND Immediate Shifted D-form**

\[
\text{andis. } RA,RS,UI
\]

\[
\begin{array}{cccccc}
29 & RS & RA & UI & & 31 \\
\hline
0 & 6 & 11 & 16 & & 31 \\
\end{array}
\]

\[RA \leftarrow (RS) \& (320 || UI || 160)\]

The contents of register \(RS\) are ANDed with \(320 || UI || 160\) and the result is placed into register \(RA\).

**Special Registers Altered:**

\(CR0\)

**OR Immediate D-form**

\[
\text{ori } RA,RS,UI
\]

\[
\begin{array}{cccccc}
24 & RS & RA & UI & & 31 \\
\hline
0 & 6 & 11 & 16 & & 31 \\
\end{array}
\]

\[RA \leftarrow (RS) \| (480 || UI)\]

The contents of register \(RS\) are ORed with \(480 || UI\) and the result is placed into register \(RA\).

The preferred "no-op" (an instruction that does nothing) is:

\[\text{ori } 0,0,0\]

Some other forms of \(ori Rx,Rx,0\) provide special functions; see <xref to Book III Section 4.4.2+>.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Example of extended mnemonics for *OR Immediate*:

Extended mnemonic: \(\text{ori } 0,0,0\)

Equivalent to: \(\text{nop}\)
**OR Immediate Shifted D-form**

```
<table>
<thead>
<tr>
<th>25</th>
<th>RS</th>
<th>RA</th>
<th>UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>
```

```
RA ← (RS) | | 320 | | UI | | 160 |
```

The contents of register RS are ORed with 320 | | UI | | 160 and the result is placed into register RA.

**Special Registers Altered:**
None

**XOR Immediate D-form**

```
<table>
<thead>
<tr>
<th>26</th>
<th>RS</th>
<th>RA</th>
<th>UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>
```

```
RA ← (RS) XOR (480 | | UI)
```

The contents of register RS are XORed with 480 | | UI and the result is placed into register RA.

The executed form of a "no-op" (an instruction that does nothing, but consumes execution resources nevertheless) is:

```
xori 0,0,0
```

**Special Registers Altered:**
None

**Extended Mnemonics:**

Example of extended mnemonics for XOR Immediate:

```
<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>xnor</td>
<td>xori 0,0,0</td>
</tr>
</tbody>
</table>
```

**Programming Note**

The executed form of no-op should be used only when the intent is to alter the timing of a program.

**OR Immediate Shifted D-form**

```
<table>
<thead>
<tr>
<th>27</th>
<th>RS</th>
<th>RA</th>
<th>UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>
```

```
RA ← (RS) XOR | | 320 | | UI | | 160 |
```

The contents of register RS are XORed with 320 | | UI | | 160 and the result is placed into register RA.

**Special Registers Altered:**
None

**AND X-form**

```
and   RA,RS,RB (Rc=0)
and.  RA,RS,RB (Rc=1)
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

```
RA ← (RS) & (RB)
```

The contents of register RS are ANDed with the contents of register RB and the result is placed into register RA.

**Special Registers Altered:**
None

**XOR X-form**

```
xor   RA,RS,RB (Rc=0)
xor.  RA,RS,RB (Rc=1)
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>316</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

```
RA ← (RS) ⊕ (RB)
```

The contents of register RS are XORed with the contents of register RB and the result is placed into register RA.

**Special Registers Altered:**
None

**NAND X-form**

```
nand  RA,RS,RB (Rc=0)
nand. RA,RS,RB (Rc=1)
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>476</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

```
RA ← ¬((RS) & (RB))
```

The contents of register RS are ANDed with the contents of register RB and the complemented result is placed into register RA.

**Special Registers Altered:**
CR0 (if Rc=1)

**Programming Note**

nand or nor with RS=RB can be used to obtain the one’s complement.
**OR X-form**

Or

\[
\text{or. } \text{RA,RS,RB} \quad \text{(Rc=0)}
\]

Or.

\[
\text{or. } \text{RA,RS,RB} \quad \text{(Rc=1)}
\]

\[
\begin{array}{cccccc}
31 & RS & RA & RB & 444 & R6 \\
0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[RA \leftarrow (RS) \mid (RB)\]

The contents of register RS are ORed with the contents of register RB and the result is placed into register RA.

Some forms of or Rx,Rx,Rx provide special functions; see Section 3.2 and Section 4.3.3, both in Book II.

**Special Registers Altered:**

CRO

(if Rc=1)

**Extended Mnemonics:**

Example of extended mnemonics for OR:

Extended mnemonic:  
Equivalent to:

\[
m_r \quad Rx, Ry \quad \text{or} \quad Rx, Ry, Ry
\]

**OR with Complement X-form**

orc

\[
\text{orc. } \text{RA,RS,RB} \quad \text{(Rc=0)}
\]

orc.

\[
\text{orc. } \text{RA,RS,RB} \quad \text{(Rc=1)}
\]

\[
\begin{array}{cccccc}
31 & RS & RA & RB & 412 & R6 \\
0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[RA \leftarrow (RS) \mid \neg(RB)\]

The contents of register RS are ORed with the complement of the contents of register RB and the result is placed into register RA.

**Special Registers Altered:**

CRO

(if Rc=1)

**NOR X-form**

nor

\[
\text{nor. } \text{RA,RS,RB} \quad \text{(Rc=0)}
\]

nor.

\[
\text{nor. } \text{RA,RS,RB} \quad \text{(Rc=1)}
\]

\[
\begin{array}{cccccc}
31 & RS & RA & RB & 124 & R6 \\
1 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[RA \leftarrow \neg((RS) \mid (RB))\]

The contents of register RS are ORed with the contents of register RB and the complemented result is placed into register RA.

**Special Registers Altered:**

CRO

(if Rc=1)

**Equivalent X-form**

eqv

\[
\text{eqv. } \text{RA,RS,RB} \quad \text{(Rc=0)}
\]

eqv.

\[
\text{eqv. } \text{RA,RS,RB} \quad \text{(Rc=1)}
\]

\[
\begin{array}{cccccc}
31 & RS & RA & RB & 284 & R6 \\
0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[RA \leftarrow (RS) \equiv (RB)\]

The contents of register RS are XORed with the contents of register RB and the complemented result is placed into register RA.

**Special Registers Altered:**

CRO

(if Rc=1)

**AND with Complement X-form**

andc

\[
\text{andc. } \text{RA,RS,RB} \quad \text{(Rc=0)}
\]

andc.

\[
\text{andc. } \text{RA,RS,RB} \quad \text{(Rc=1)}
\]

\[
\begin{array}{cccccc}
31 & RS & RA & RB & 60 & R6 \\
0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[RA \leftarrow (RS) \& \neg(RB)\]

The contents of register RS are ANDed with the complement of the contents of register RB and the result is placed into register RA.

**Special Registers Altered:**

CRO

(if Rc=1)
### Extend Sign Byte X-form

**extsb**
```
RA,RS  (Rc=0)
```

**extsb.**
```
RA,RS  (Rc=1)
```

1. \( s \leftarrow (RS)_{56} \)
2. \( RA_{56:63} \leftarrow (RS)_{56:63} \)
3. \( RA_{0:55} \leftarrow s_{56} \)

\( (RS)_{56:63} \) are placed into \( RA_{56:63} \), \( RA_{0:55} \) are filled with a copy of \( (RS)_{56} \).

**Special Registers Altered:**

- \( CR0 \) (if \( Rc=1 \))

### Extend Sign Halfword X-form

**extsh**
```
RA,RS  (Rc=0)
```

**extsh.**
```
RA,RS  (Rc=1)
```

1. \( s \leftarrow (RS)_{48} \)
2. \( RA_{48:63} \leftarrow (RS)_{48:63} \)
3. \( RA_{0:47} \leftarrow s_{48} \)

\( (RS)_{48:63} \) are placed into \( RA_{48:63} \), \( RA_{0:47} \) are filled with a copy of \( (RS)_{48} \).

**Special Registers Altered:**

- \( CR0 \) (if \( Rc=1 \))

### Compare Bytes X-form

**cmpb**
```
RA,RS,RB
```

```text
do n = 0 to 7
  if \( RS_{8 \times n:8 \times (n+7)} = (RB)_{8 \times n:8 \times (n+7)} \) then
    RA_{8 \times n:8 \times (n+7)} \leftarrow 0xFF
  else
    RA_{8 \times n:8 \times (n+7)} \leftarrow 0x00
```

Each byte of the contents of register \( RS \) is compared to each corresponding byte of the contents in register \( RB \). If they are equal, the corresponding byte in \( RA \) is set to \( 0xFF \). Otherwise the corresponding byte in \( RA \) is set to \( 0x00 \).

**Special Registers Altered:**

- None

---

### Count Leading Zeros Word X-form

**cntlzwr**
```
RA,RS  (Rc=0)
```

**cntlzwr.**
```
RA,RS  (Rc=1)
```

1. \( n \leftarrow 32 \)
2. **do while** \( n < 64 \)
   - **if** \( (RS)_{63-n} = 0b1 \) then leave
   - \( n \leftarrow n + 1 \)
3. \( RA \leftarrow n - 32 \)

A count of the number of consecutive zero bits starting at bit 32 of register \( RS \) is placed into register \( RA \). This number ranges from 0 to 32, inclusive.

**Special Registers Altered:**

- \( CR0 \) (if \( Rc=1 \))

---

### Programming Note

For both **Count Leading Zeros** instructions, if \( Rc=1 \) then \( LT \) is set to 0 in CR Field 0.

---

### Count Trailing Zeros Word X-form

**cnttzw**
```
RA,RS  (Rc=0)
```

**cnttzw.**
```
RA,RS  (Rc=1)
```

1. \( n \leftarrow 0 \)
2. **do while** \( n < 32 \)
   - **if** \( (RS)_{64-n} = 0b01 \) then leave
   - \( n \leftarrow n + 1 \)
3. \( RA \leftarrow EXTZ64(n) \)

A count of the number of consecutive zero bits starting at bit 63 of the rightmost word of register \( RS \) is placed into register \( RA \). This number ranges from 0 to 32, inclusive.

**Special Registers Altered:**

- \( CR0 \) (if \( Rc=1 \))

---

102 Power ISA™ I
**Population Count Bytes X-form**

\[ \text{popcntb} \quad \text{RA, RS} \]

\[
\begin{array}{cccccc}
31 & RS & RA & /// & 122 & Rb \\
0 & 6 & 11 & 16 & 21 & 31
\end{array}
\]

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 7 \\
& \quad n \leftarrow 0 \\
\text{do } j &= 0 \text{ to } 7 \\
& \quad \text{if } (RS)(i \times 8)+j = 1 \text{ then} \\
& \quad n \leftarrow n+1 \\
RA(i \times 8):(i \times 8)+7 & \leftarrow n
\end{align*}
\]

A count of the number of one bits in each byte of register \( RS \) is placed into the corresponding byte of register \( RA \). This number ranges from 0 to 8, inclusive.

**Special Registers Altered:**

None

**Population Count Words X-form**

\[ \text{popcntw} \quad \text{RA, RS} \]

\[
\begin{array}{cccccc}
31 & RS & RA & /// & 378 & / \\
0 & 6 & 11 & 16 & 21 & 31
\end{array}
\]

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 1 \\
& \quad n \leftarrow 0 \\
\text{do } j &= 0 \text{ to } 31 \\
& \quad \text{if } (RS)(i \times 32)+j = 1 \text{ then} \\
& \quad n \leftarrow n+1 \\
RA(i \times 32):(i \times 32)+31 & \leftarrow n
\end{align*}
\]

A count of the number of one bits in each word of register \( RS \) is placed into the corresponding word of register \( RA \). This number ranges from 0 to 32, inclusive.

**Special Registers Altered:**

None

**Parity Word X-form**

\[ \text{prtyw} \quad \text{RA, RS} \]

\[
\begin{array}{cccccc}
31 & RS & RA & /// & 154 & / \\
0 & 6 & 11 & 16 & 21 & 31
\end{array}
\]

\[
\begin{align*}
s & \leftarrow 0 \\
t & \leftarrow 0 \\
\text{do } i &= 0 \text{ to } 3 \\
& \quad s \leftarrow s \oplus (RS)(i \times 8)+7 \\
& \quad t \leftarrow t \oplus (RS)(i \times 8)+7 \\
RA0:31 & \leftarrow 320 || s \\
RA32:63 & \leftarrow 320 || t
\end{align*}
\]

The least significant bit in each byte of \( (RS)0:31 \) is examined. If there is an odd number of one bits the value 1 is placed into \( RA0:31 \); otherwise the value 0 is placed into \( RA0:31 \). The least significant bit in each byte of \( (RS)32:63 \) is examined. If there is an odd number of one bits the value 1 is placed into \( RA32:63 \); otherwise the value 0 is placed into \( RA32:63 \).

**Special Registers Altered:**

None

**Programming Note**

The **Parity** instructions are designed to be used in conjunction with the **Population Count** instruction to compute the parity of words or a doubleword. The parity of the upper and lower words in \( (RS) \) can be computed as follows.

\[ \text{popcntb} \quad \text{RA, RS} \]
\[ \text{prtyw} \quad \text{RA, RA} \]

The parity of \( (RS) \) can be computed as follows.

\[ \text{popcntb} \quad \text{RA, RS} \]
\[ \text{prtyd} \quad \text{RA, RA} \]
3.3.13.1 64-bit Fixed-Point Logical Instructions

**Extend Sign Word X-form**

\[
\text{extsw} \quad RA, RS \quad (Rc = 0)
\]

\[
\text{extsw.} \quad RA, RS \quad (Rc = 1)
\]

\[
\begin{array}{cccccc}
31 & RS & RA & \text{///} & 16 & 21
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 6 & 11 & 16 & 21 & 986
\end{array}
\]

\[
\begin{align*}
s & \leftarrow (RS)_{32} \\
R_{32:63} & \leftarrow (RS)_{32:63} \\
R_{6:32} & \leftarrow 32
\end{align*}
\]

The sign word \((RS)_{32:63}\) is placed into \(R_{32:63}\). \(R_{6:32}\) are filled with a copy of \((RS)_{32}\).

**Special Registers Altered:**

CR0 (if \(Rc = 1\))

**Population Count Doubleword X-form**

\[
\text{popcntd} \quad RA, RS
\]

\[
\begin{array}{cccccc}
31 & RS & RA & \text{///} & 16 & 21
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 6 & 11 & 16 & 21 & 506
\end{array}
\]

\[
\begin{align*}
n & \leftarrow 0 \\
\text{do i = 0 to 63} & \text{ then} \\
\text{if } (RS)_i = 1 & \text{ then} \\
\quad n & \leftarrow n + 1 \\
\quad RA & \leftarrow n
\end{align*}
\]

A count of the number of one bits in register \(RS\) is placed into register \(RA\). This number ranges from 0 to 64, inclusive.

**Special Registers Altered:**

None

**Parity Doubleword X-form**

\[
\text{prtyd} \quad RA, RS
\]

\[
\begin{array}{cccccc}
31 & RS & RA & \text{///} & 16 & 21
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 6 & 11 & 16 & 21 & 186
\end{array}
\]

\[
\begin{align*}
s & \leftarrow 0 \\
\text{do } i = 0 \text{ to 7} & \\
\text{if } (RS)_{63-i} = 1 & \text{ then} \\
\quad s & \leftarrow s \oplus (RS)_{63-i} \\
\quad RA & \leftarrow \text{EXTZ64}(n)
\end{align*}
\]

The least significant bit in each byte of the contents of register \(RS\) is examined. If there is an odd number of one bits the value 1 is placed into register \(RA\); otherwise the value 0 is placed into register \(RA\).

**Special Registers Altered:**

None

**Count Leading Zeros Doubleword X-form**

\[
\text{cntlzd} \quad RA, RS \quad (Rc = 0)
\]

\[
\text{cntlzd.} \quad RA, RS \quad (Rc = 1)
\]

\[
\begin{array}{cccccc}
31 & RS & RA & \text{///} & 16 & 21
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 6 & 11 & 16 & 21 & 58
\end{array}
\]

\[
\begin{align*}
n & \leftarrow 0 \\
\text{do while } n < 64 & \text{ then} \\
\text{if } (RS)_n = 1 & \text{ then leave} \\
\quad n & \leftarrow n + 1 \\
\quad RA & \leftarrow n
\end{align*}
\]

A count of the number of consecutive zero bits starting at bit 0 of register \(RS\) is placed into register \(RA\). This number ranges from 0 to 64, inclusive.

If \(Rc = 1\), CR Field 0 is set to reflect the result.

**Special Registers Altered:**

CR0 (if \(Rc = 1\))

**Count Trailing Zeros Doubleword X-form**

\[
\text{cnttzd} \quad RA, RS \quad (Rc = 0)
\]

\[
\text{cnttzd.} \quad RA, RS \quad (Rc = 1)
\]

\[
\begin{array}{cccccc}
31 & RS & RA & \text{///} & 16 & 21
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 6 & 11 & 16 & 21 & 570
\end{array}
\]

\[
\begin{align*}
n & \leftarrow 0 \\
\text{do while } n < 64 & \text{ then} \\
\text{if } (RS)_{63-n} = 0b1 & \text{ then leave} \\
\quad n & \leftarrow n + 1 \\
\quad RA & \leftarrow \text{EXTZ64}(n)
\end{align*}
\]

A count of the number of consecutive zero bits starting at bit 63 of register \(RS\) is placed into register \(RA\). This number ranges from 0 to 64, inclusive.

If \(Rc\) is equal to 1, CR Field 0 is set to reflect the result.

**Special Registers Altered:**

None
Count Leading Zeros Doubleword under bit Mask X-form

```plaintext
cntzdm  RA,RS,RB
      31  RS  RA  RB  59 / 31 6 11 16 21
```

count = 0
do i = 0 to 63
  if((RB)i=1) then do
    if((RS)i=1) then break
    count ← count + 1
  end
end
RA ← EXTZ64(count)

Let \( n \) be the number of bits in register RB having the value 1.

Extract and pack together the contents of the bits in register RS corresponding to a mask specified in register RB, creating an \( n \)-bit value.

Count the number of contiguous leftmost 0 bits in the \( n \)-bit extracted value and place the result into register RA.

Special Registers Altered:
None

Count Trailing Zeros Doubleword under bit Mask X-form

```plaintext
cntzdm  RA,RS,RB
      31  RS  RA  RB  571 / 31 6 11 16 21
```

count = 0
do i = 0 to 63
  if((RB)63-i=1) then do
    if((RS)63-i=1) then break
    count ← count + 1
  end
end
RA ← EXTZ64(count)

Let \( n \) be the number of bits in register RB having the value 1.

Extract and pack together the contents of bits in register RS corresponding to a mask specified in register RB, creating an \( n \)-bit value.

Count the number of contiguous rightmost 0 bits in the \( n \)-bit extracted value and place the result into register RA.

Special Registers Altered:
None

Bit Permute Doubleword X-form

```plaintext
bpermd  RA,RS,RB
      31  RS  RA  RB  252 / 31 6 11 16 21
```

Eight permuted bits are produced. For each permuted bit \( i \) where \( i \) ranges from 0 to 7 and for each byte \( i \) of RS, do the following.

If byte \( i \) of RS is less than 64, permuted bit \( i \) is set to the bit of RB specified by byte \( i \) of RS; otherwise permuted bit \( i \) is set to 0.

The permuted bits are placed in the least-significant byte of RA, and the remaining bits are filled with 0s.

Special Registers Altered:
None

Programming Note

The fact that the permuted bit is 0 if the corresponding index value exceeds 63 permits the permuted bits to be selected from a 128-bit quantity, using a single index register. For example, assume that the 128-bit quantity \( Q \), from which the permuted bits are to be selected, is in registers r2 (high-order 64 bits of \( Q \)) and r3 (low-order 64 bits of \( Q \)), that the index values are in register r1, with each byte of r1 containing a value in the range 0:127, and that each byte of register r4 contains the value 64. The following code sequence selects eight permuted bits from \( Q \) and places them into the low-order byte of r6.

```plaintext
bpermd  r6,r1,r2 # select from high-order half of Q
xor  r0,r1,r4 # adjust index values
bpermd  r5,r0,r3 # select from low-order half of Q
or  r6,r6,r5 # merge the two selections
```
**Centrifuge Doubleword X-form**

```plaintext
cfuged RA,RS,RB
```

```plaintext
ptr0 ← 0
ptr1 ← 0
do i = 0 to 63
  if((RB)i==0) then do
    resultptr0 ← (RS)i
    ptr0 ← ptr0 + 1
  end
  if((RB)63-i==1) then do
    result63-ptr1 ← (RS)63-i
    ptr1 ← ptr1 + 1
  end
end
RA ← result
```

The bits in GPR[RS] whose corresponding bits in the mask in GPR[RB] equal 1 are placed in the rightmost bits in GPR[RA] maintaining their relative original order. The other bits in GPR[RS] are placed in the leftmost bits in GPR[RA] maintaining their relative original order.

**Special Registers Altered:**
None

**Parallel Bits Extract Doubleword X-form**

```plaintext
pextd RA,RS,RB
```

```plaintext
result ← 0
mask ← (RB)
m ← 0
k ← 0
do while(m < 64)
  if((RB)63-m == 1) then do
    result63-m ← (RS)63-m
    k ← k + 1
  end
  m ← m + 1
end
RA ← result
```

Let mask be the contents of register RB.

The contents of the bits in register RS corresponding to bits in mask containing a 1 are packed into an n-bit value. The extracted value is placed into register RA.

**Special Registers Altered:**
None

---

**Parallel Bits Deposit Doubleword X-form**

```plaintext
pdepd RA,RS,RB
```

```plaintext
result ← 0
mask ← (RB)
m ← 0
k ← 0
do while(m < 64)
  if((RB)63-m == 1) then do
    result63-m ← (RS)63-m
    k ← k + 1
  end
  m ← m + 1
end
RA ← result
```

Let mask be the contents of register RB.

Let n be the number of bits in mask having the value 1.

The contents of the rightmost n bits of register RS are placed into register RA under control of mask as follows.

- The contents of bit 63 of register RS are placed into the bit in register RA corresponding to the rightmost bit in mask that contains a 1,
- The contents of bit 62 of register RS are placed into the bit in register RA corresponding to the second rightmost bit in mask that contains a 1, and so forth until
- The contents of bit 64-n of register RS are placed into the bit in register RA corresponding to the leftmost bit in mask that contains a 1.

The contents of bits in register RA corresponding to bits in mask that contain a 0 are set to 0.

**Special Registers Altered:**
None
3.3.14 Fixed-Point Rotate and Shift Instructions

The Fixed-Point Facility performs rotation operations on data from a GPR and returns the result, or a portion of the result, to a GPR.

The rotation operations rotate a 64-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 63.

Two types of rotation operation are supported.

For the first type, denoted rotatem64 or ROTL64, the value rotated is the given 64-bit value. The rotatem64 operation is used to rotate a given 64-bit quantity.

For the second type, denoted rotatem32 or ROTL32, the value rotated consists of two copies of bits 32:63 of the given 64-bit value, one copy in bits 0:31 and the other in bits 32:63. The rotatem32 operation is used to rotate a given 32-bit quantity.

The Rotate and Shift instructions employ a mask generator. The mask is 64 bits long, and consists of 1-bits from a start bit, mstart, through and including a stop bit, mstop, and 0-bits elsewhere. The values of mstart and mstop range from 0 to 63. If mstart > mstop, the 1-bits wrap around from position 63 to position 0. Thus the mask is formed as follows:

```plaintext
if mstart ≤ mstop then
  maskmstart:mstop = ones
  maskall other bits = zeros
else
  maskmstart:63 = ones
  mask0:mstop = ones
  maskall other bits = zeros
```

There is no way to specify an all-zero mask.

For instructions that use the rotatem32 operation, the mask start and stop positions are always in the low-order 32 bits of the mask.

The use of the mask is described in following sections.

The Rotate and Shift instructions with Rc=1 set the first three bits of CR field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions” on page 74. Rotate and Shift instructions do not change the OV, OV32, and SO bits. Rotate and Shift instructions, except algebraic right shifts, do not change the CA and CA32 bits.

Extended mnemonics for rotates and shifts

The Rotate and Shift instructions, while powerful, can be complicated to code (they have up to five operands). A set of extended mnemonics is provided that allow simpler coding of often-used functions such as clearing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and performing simple rotates and shifts. Some of these are shown as examples with the Rotate instructions. See Appendix C, “Assembler Extended Mnemonics” on page 1011 for additional extended mnemonics.

### 3.3.14.1 Fixed-Point Rotate Instructions

These instructions rotate the contents of a register. The result of the rotation is

- inserted into the target register under control of a mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register remains unchanged); or
- ANDed with a mask before being placed into the target register.

The Rotate Left instructions allow right-rotation of the contents of a register to be performed (in concept) by a left-rotation of 64 - n, where n is the number of bits by which to rotate right. They allow right-rotation of the contents of the low-order 32 bits of a register to be performed (in concept) by a left-rotation of 32 - n, where n is the number of bits by which to rotate right.

<table>
<thead>
<tr>
<th>Rotate Left Word Immediate then AND with Mask M-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>nlwimm</td>
</tr>
<tr>
<td>nlwimm.</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>n ← SH</td>
</tr>
<tr>
<td>r ← ROTL32(RS),m32, n)</td>
</tr>
<tr>
<td>m ← MASK(MB+32, ME+32)</td>
</tr>
<tr>
<td>RA ← r &amp; m</td>
</tr>
</tbody>
</table>

The contents of register RS are rotated32 left SH bits. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

- CR0 (if Rc=1)
Extended Mnemonics:

Examples of extended mnemonics for Rotate Left Word Immediate then AND with Mask:

<table>
<thead>
<tr>
<th>Extended mnemonic:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>extl/w Rx,Ry,n,b</td>
<td>rlwnm Rx,Ry,b,0,n-1</td>
</tr>
<tr>
<td>srwi Rx,Ry,n</td>
<td>rlwnm Rx,Ry,32-n,n,31</td>
</tr>
<tr>
<td>clrrw Rx,Ry,n</td>
<td>rlwnm Rx,Ry,0,0,31-n</td>
</tr>
</tbody>
</table>

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

*rlwinm* can be used to extract an n-bit field that starts at bit position b in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting SH=b+n, MB=32-n, and ME=n-1. It can be used to extract an n-bit field that starts at bit position b in RSL, left-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting SH=b, MB=0, and ME=n-1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by n bits, by setting SH=n (32-n), MB=0, and ME=31. It can be used to shift the contents of the low-order 32 bits of a register right by n bits, by setting SH=32-n, MB=n, and ME=31. It can be used to clear the high-order b bits of the low-order 32 bits of a register and then shift the result left by n bits, by setting SH=n, MB=b-n, and ME=31-n. It can be used to clear the low-order n bits of the low-order 32 bits of a register, by setting SH=0, MB=0, and ME=31-n.

For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for some of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 1011.

**Rotate Left Word Immediate then Mask Insert M-form**

\[
\text{rlwinm} \quad (Rc=0) \\
\text{rlwinm} \quad (Rc=1)
\]

Extended Mnemonics:

Example of extended mnemonics for Rotate Left Word Immediate then Mask Insert:

<table>
<thead>
<tr>
<th>Extended mnemonic:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>inslwi Rx,Ry,n,b</td>
<td>rlwinm Rx,Ry,0,3</td>
</tr>
</tbody>
</table>
Let $RA_L$ represent the low-order 32 bits of register $RA$, with the bits numbered from 0 through 31.

$rhwimi$ can be used to insert an $n$-bit field that is left-justified in the low-order 32 bits of register $RS$, into $RA_L$ starting at bit position $b$, by setting $SH=32-b$, $MB=b$, and $ME=(b+n)-1$. It can be used to insert an $n$-bit field that is right-justified in the low-order 32 bits of register $RS$, into $RA_L$ starting at bit position $b$, by setting $SH=32-(b+n)$, $MB=b$, and $ME=(b+n)-1$.

Extended mnemonics are provided for both of these uses; see Appendix C, "Assembler Extended Mnemonics" on page 1011.
3.3.14.1.1 64-bit Fixed-Point Rotate Instructions

**Rotate Left Doubleword Immediate then Clear Left MD-form**

\[
\text{rdicl} \quad \text{RA}, \text{RS}, \text{SH}, \text{MB} \quad (\text{Rc}=0)
\]

\[
\text{rdicl.} \quad \text{RA}, \text{RS}, \text{SH}, \text{MB} \quad (\text{Rc}=1)
\]

\[
\begin{array}{ccccccc}
0 & 30 & 6 & 11 & 16 & 21 & 27 & 30 31 \\
\end{array}
\]

\[
n \leftarrow \text{sh}_5 || \text{sh}_0:4
\]

\[
r \leftarrow \text{ROTL}_{64}(\text{RS}), n
\]

\[
b \leftarrow \text{mb}_5 || \text{mb}_0:4
\]

\[
m \leftarrow \text{MASK}(b, 63)
\]

\[
\text{RA} \leftarrow r \& m
\]

The contents of register RS are rotated \( 64 \) left \( \text{SH} \) bits. A mask is generated having 1-bits from bit \( \text{MB} \) through bit 63 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

**Special Registers Altered:**

CR0 (if \( \text{Rc}=1 \))

**Extended Mnemonics:**

Examples of extended mnemonics for **Rotate Left Doubleword Immediate then Clear Left**:

<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>extrdi Rx, Ry, n, b</td>
<td>rldicl Rx, Ry, b+n, 64-n</td>
</tr>
<tr>
<td>srdi Rx, Ry, n</td>
<td>rldicl Rx, Ry, 64-n, n</td>
</tr>
<tr>
<td>clrldi Rx, Ry, n</td>
<td>rldicl Rx, Ry, 0, n</td>
</tr>
</tbody>
</table>

**Programming Note**

\textit{rdicl} can be used to extract an \( n \)-bit field that starts at bit position \( b \) in register RS, right-justified into register RA (clearing the remaining \( 64-n \) bits of RA), by setting \( SH=b+n \) and \( MB=64-n \). It can be used to rotate the contents of a register left (right) by \( n \) bits, by setting \( SH=n \) (64-\( n \)) and \( MB=0 \). It can be used to shift the contents of a register right by \( n \) bits, by setting \( SH=64-n \) and \( MB=n \). It can be used to clear the high-order \( n \) bits of a register, by setting \( SH=0 \) and \( MB=n \).

Extended mnemonics are provided for all of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 1011.

**Rotate Left Doubleword Immediate then Clear Right MD-form**

\[
\text{rdicr} \quad \text{RA}, \text{RS}, \text{SH}, \text{ME} \quad (\text{Rc}=0)
\]

\[
\text{rdicr.} \quad \text{RA}, \text{RS}, \text{SH}, \text{ME} \quad (\text{Rc}=1)
\]

\[
\begin{array}{ccccccc}
0 & 30 & 6 & 11 & 16 & 21 & 27 & 30 31 \\
\end{array}
\]

\[
n \leftarrow \text{sh}_5 || \text{sh}_0:4
\]

\[
r \leftarrow \text{ROTL}_{64}(\text{RS}), n
\]

\[
e \leftarrow \text{me}_5 || \text{me}_0:4
\]

\[
m \leftarrow \text{MASK}(0, e)
\]

\[
\text{RA} \leftarrow r \& m
\]

The contents of register RS are rotated \( 64 \) left \( \text{SH} \) bits. A mask is generated having 1-bits from bit 0 through bit \( \text{ME} \) and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

**Special Registers Altered:**

CR0 (if \( \text{Rc}=1 \))

**Extended Mnemonics:**

Examples of extended mnemonics for **Rotate Left Doubleword Immediate then Clear Right**:

<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>extrdi Rx, Ry, n, b</td>
<td>rldicr Rx, Ry, b, n-1</td>
</tr>
<tr>
<td>sldi Rx, Ry, n</td>
<td>rldicr Rx, Ry, 63-n</td>
</tr>
<tr>
<td>crldi Rx, Ry, n</td>
<td>rldicr Rx, Ry, 63-n</td>
</tr>
</tbody>
</table>

**Programming Note**

\textit{rdicr} can be used to extract an \( n \)-bit field that starts at bit position \( b \) in register RS, left-justified into register RA (clearing the remaining \( 64-n \) bits of RA), by setting \( SH=b \) and \( ME=n-1 \). It can be used to rotate the contents of a register left (right) by \( n \) bits, by setting \( SH=n \) (64-\( n \)) and \( ME=63 \). It can be used to shift the contents of a register left by \( n \) bits, by setting \( SH=n \) and \( ME=63-n \). It can be used to clear the low-order \( n \) bits of a register, by setting \( SH=0 \) and \( ME=63-n \).

Extended mnemonics are provided for all of these uses (some devolve to \textit{rdicl}); see Appendix C, “Assembler Extended Mnemonics” on page 1011.
**Rotate Left Doubleword Immediate then Clear**

**MD-form**

```
   rldic. RA,RS,SH,MB (Rc=0)
   rldc. RA,RS,SH,MB (Rc=1)
```

<table>
<thead>
<tr>
<th>30</th>
<th>31</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>RA</td>
<td>sh</td>
<td>mb</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>21</td>
<td>27</td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
n ← sh5 || sh0:4
r ← NOTL64(RS), n
b ← mb5 || mb0:4
m ← MASK(b, ¬n)
RA ← r & m
```

The contents of register $RS$ are rotated left the number of bits specified by $(RB)_{58:63}$. A mask is generated having 1-bits from bit $MB$ through bit $63 \cdot SH$ and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register $RA$.

**Special Registers Altered:**

$CR0$ (if $Rc=1$)

**Extended Mnemonics:**

Example of extended mnemonics for *Rotate Left Doubleword Immediate then Clear:*

```
crlslidl Rx,Ry,b,n rldc Rx,Ry,n,b-n
```

**Programming Note**

`rldic` can be used to clear the high-order $b$ bits of the contents of a register and then shift the result left by $n$ bits, by setting $SH=n$ and $MB=b \cdot n$. It can be used to clear the high-order $n$ bits of a register, by setting $SH=0$ and $MB=n$.

Extended mnemonics are provided for both of these uses (the second devolves to `rldicl`); see Appendix C, “Assembler Extended Mnemonics” on page 1011.

**Rotate Left Doubleword then Clear Left**

**MDS-form**

```
   rldcl RA,RS,RB,MB (Rc=0)
   rldcl. RA,RS,RB,MB (Rc=1)
```

<table>
<thead>
<tr>
<th>30</th>
<th>31</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>RA</td>
<td>RB</td>
<td>mb</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>21</td>
<td>27</td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
n ← (RB)_{58:63}
r ← NOTL64(RS), n
b ← mb5 || mb0:4
m ← MASK(b, 63)
RA ← r & m
```

The contents of register $RS$ are rotated left the number of bits specified by $(RB)_{58:63}$. A mask is generated having 1-bits from bit $MB$ through bit 63 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register $RA$.

**Special Registers Altered:**

$CR0$ (if $Rc=1$)

**Extended Mnemonics:**

Example of extended mnemonics for *Rotate Left Doubleword then Clear Left:*

```
rcpld Rx,Ry,Rz rldcl Rx,Ry,Rz,0
```

**Programming Note**

`rldcl` can be used to extract an $n$-bit field that starts at variable bit position $b$ in register $RS$, right-justified into register $RA$ (clearing the remaining $64-n$ bits of $RA$), by setting $RB_{58:63}=b+n$ and $MB=64-n$. It can be used to rotate the contents of a register left (right) by variable $n$ bits, by setting $RB_{58:63}=n$ ($64-n$) and $MB=0$.

Extended mnemonics are provided for some of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 1011.
**Rotate Left Doubleword then Clear Right MDS-form**

\[
\begin{align*}
\text{rldcr} & \quad \text{RA,RS,RB,ME} \\
\text{rldcr} & \quad \text{RA,RS,RB,ME}
\end{align*}
\]

\[
\begin{array}{cccccc}
0 & 30 & 6 & 11 & 16 & 21 & 27 & 31 \\
\end{array}
\]

\[
\begin{array}{l}
\begin{align*}
n & \leftarrow (RB)_{58:63} \\
r & \leftarrow \text{ROTL}_{64}((RS), n) \\
e & \leftarrow \text{me}_5 || \text{me}_0:4 \\
m & \leftarrow \text{MASK}(0, e) \\
\text{RA} & \leftarrow r \& m
\end{align*}
\end{array}
\]

The contents of register \( RS \) are rotated left the number of bits specified by \( (RB)_{58:63} \). A mask is generated having 1-bits from bit \( ME \) and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register \( RA \).

**Special Registers Altered:**

- CR0 (if \( Rc=1 \))

---

**Programming Note**

**rldcr** can be used to extract an \( n \)-bit field that starts at variable bit position \( b \) in register \( RS \), left-justified into register \( RA \) (clearing the remaining \( 64-n \) bits of \( RA \)), by setting \( RB_{58:63}=b \) and \( ME=n-1 \). It can be used to rotate the contents of a register left (right) by variable \( n \) bits, by setting \( RB_{58:63}=n \) \( (64-n) \) and \( ME=63 \).

Extended mnemonics are provided for some of these uses (some devolve to **rldcl**); see Appendix C, "Assembler Extended Mnemonics" on page 1011.

---

**Rotate Left Doubleword Immediate then Mask Insert MD-form**

\[
\begin{align*}
\text{rldimi} & \quad \text{RA,RS,SH,MB} \\
\text{rldimi} & \quad \text{RA,RS,SH,MB}
\end{align*}
\]

\[
\begin{array}{cccccc}
0 & 30 & 6 & 11 & 16 & 21 & 27 & 31 \\
\end{array}
\]

\[
\begin{array}{l}
\begin{align*}
n & \leftarrow \text{sh}_5 || \text{sh}_0:4 \\
r & \leftarrow \text{ROTL}_{64}((RS), n) \\
b & \leftarrow \text{mb}_5 || \text{mb}_0:4 \\
m & \leftarrow \text{MASK}(b, \neg n) \\
\text{RA} & \leftarrow r \& m | (RA) \& \neg m
\end{align*}
\end{array}
\]

The contents of register \( RS \) are rotated left \( SH \) bits. A mask is generated having 1-bits from bit \( MB \) through bit \( 63-SH \) and 0-bits elsewhere. The rotated data are inserted into register \( RA \) under control of the generated mask.

**Special Registers Altered:**

- CR0 (if \( Rc=1 \))

---

**Extended Mnemonics:**

Example of extended mnemonics for **Rotate Left Doubleword Immediate then Mask Insert**:

- Extended mnemonic: **insrdi** \( Rx,Ry,n,b \) equivalent to: **rldimi** \( Rx,Ry,64-(b+n),b \)

---

**Programming Note**

**rldimi** can be used to insert an \( n \)-bit field that is right-justified in register \( RS \), into register \( RA \) starting at bit position \( b \), by setting \( SH=64-(b+n) \) and \( MB=b \).

An extended mnemonic is provided for this use; see Appendix C, "Assembler Extended Mnemonics" on page 1011.
3.3.14.2 Fixed-Point Shift Instructions

The instructions in this section perform left and right shifts.

Extended mnemonics for shifts

Immediate-form logical (unsigned) shift operations are obtained by specifying appropriate masks and shift values for certain Rotate instructions. A set of extended mnemonics is provided to make coding of such shifts simpler and easier to understand. Some of these are shown as examples with the Rotate instructions. See Appendix C, “Assembler Extended Mnemonics” on page 1011 for additional extended mnemonics.

Programming Note

Any Shift Right Algebraic instruction, followed by addze, can be used to divide quickly by 2^n. The setting of the CA and CA32 bits by the Shift Right Algebraic instructions is independent of mode.

**Shift Left Word X-form**

```
slw RA, RS, RB  (Rc=0)
slw. RA, RS, RB  (Rc=1)
```

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>16</th>
<th>21</th>
<th>24</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>Rd</td>
</tr>
</tbody>
</table>

\[
n \leftarrow (RB)_{59:63}
\]

\[
r \leftarrow \text{ROTL32}(RS)_{32:63}, n
\]

\[
\text{if } (RB)_{58} = 0 \text{ then }
\]

\[
m \leftarrow \text{MASK}(32, 63-n)
\]

\[
\text{else } m \leftarrow 64
\]

RA \leftarrow r \& m

The contents of the low-order 32 bits of register RS are shifted left the number of bits specified by (RB)_{58:63}. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into RA_{32:63}. RA_{0:31} are set to zero. Shift amounts from 32 to 63 give a zero result.

**Special Registers Altered:**

C0 (if \( \text{Rc}=1 \))

**Shift Right Word X-form**

```
srw RA, RS, RB  (Rc=0)
srw. RA, RS, RB  (Rc=1)
```

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>16</th>
<th>21</th>
<th>536</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>Rd</td>
</tr>
</tbody>
</table>

\[
n \leftarrow (RB)_{59:63}
\]

\[
r \leftarrow \text{ROTL32}(RS)_{32:63}, 64-n
\]

\[
\text{if } (RB)_{58} = 0 \text{ then }
\]

\[
m \leftarrow \text{MASK}(n+32, 63)
\]

\[
\text{else } m \leftarrow 64
\]

RA \leftarrow r \& m

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by (RB)_{58:63}. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into RA_{32:63}. RA_{0:31} are set to zero. Shift amounts from 32 to 63 give a zero result.

**Special Registers Altered:**

C0 (if \( \text{Rc}=1 \))
### Shift Right Algebraic Word Immediate X-form

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>624</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>srawi</td>
<td>RA, RS, SH</td>
<td>(Rc=0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>srawi.</td>
<td>RA, RS, SH</td>
<td>(Rc=1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
n \leftarrow SH \\
 r \leftarrow \text{ROTL}_{32}((RS)_{32:63}, 64-n) \\
 m \leftarrow \text{MASK}(n+32, 63) \\
 s \leftarrow (RS)_{32} \\
 RA \leftarrow r \& m | (64 \& \neg m) \\
 \text{carry} \leftarrow s \& ((r \& \neg m)_{32:63} \neq 0) \\
 CA \leftarrow \text{carry} \\
 CA32 \leftarrow \text{carry}
\]

The contents of the low-order 32 bits of register `RS` are shifted right `SH` bits. Bits shifted out of position 63 are lost. Bit 32 of `RS` is replicated to fill the vacated positions on the left. The 32-bit result is placed into `RA_{32:63}`. Bit 32 of `RS` is replicated to fill `RA_{0:31}`. `CA` and `CA32` are set to 1 if the low-order 32 bits of `(RS)` contain a negative number and any 1-bits are shifted out of position 63; otherwise `CA` and `CA32` are set to 0. A shift amount of zero causes `RA` to receive `EXTS((RS)_{32:63})`, and `CA` and `CA32` to be set to 0.

**Special Registers Altered:**
- CA
- CA32
- CR0

(if Rc = 1)

### Shift Right Algebraic Word X-form

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>792</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>sraw</td>
<td>RA, RS, RB</td>
<td>(Rc=0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sraw.</td>
<td>RA, RS, RB</td>
<td>(Rc=1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
n \leftarrow (RB)_{59:63} \\
r \leftarrow \text{ROTL}_{32}((RS)_{32:63}, 64-n) \\
\text{if } (RB)_{58} = 0 \text{ then } m \leftarrow \text{MASK}(n+32, 63) \\
\text{else } m \leftarrow 64 \\
 s \leftarrow (RS)_{32} \\
 RA \leftarrow r \& m | (64 \& \neg m) \\
\text{carry} \leftarrow s \& ((r \& \neg m)_{32:63} \neq 0) \\
 CA \leftarrow \text{carry} \\
 CA32 \leftarrow \text{carry}
\]

The contents of the low-order 32 bits of register `RS` are shifted right the number of bits specified by `(RB)_{59:63}`. Bits shifted out of position 63 are lost. Bit 32 of `RS` is replicated to fill the vacated positions on the left. The 32-bit result is placed into `RA_{32:63}`. Bit 32 of `RS` is replicated to fill `RA_{0:31}`. `CA` and `CA32` are set to 1 if the low-order 32 bits of `(RS)` contain a negative number and any 1-bits are shifted out of position 63; otherwise `CA` and `CA32` are set to 0. A shift amount of zero causes `RA` to receive `EXTS((RS)_{32:63})`, and `CA` and `CA32` to be set to 0. Shift amounts from 32 to 63 give a result of 64 sign bits, and cause `CA` and `CA32` to receive the sign bit of `(RS)_{32:63}`.

**Special Registers Altered:**
- CA
- CA32
- CR0

(if Rc = 1)
### 3.3.14.2.1 64-bit Fixed-Point Shift Instructions

#### Shift Left Doubleword X-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Registers</th>
<th>Destination Registers</th>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sld RA,RS,RB</td>
<td>(Rc=0)</td>
<td>RA,RS,RB</td>
<td></td>
<td>64-bit fixed-point shift instruction</td>
</tr>
<tr>
<td>sld. RA,RS,RB</td>
<td>(Rc=1)</td>
<td>RA,RS,RB</td>
<td></td>
<td>64-bit fixed-point shift instruction with sign extension</td>
</tr>
</tbody>
</table>

The contents of register `RS` are shifted left the number of bits specified by `(RB)57:63`. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into register `RA`. Shift amounts from 64 to 127 give a zero result.

**Special Registers Altered:**

- `CR0` (if `Rc=1`)

#### Shift Right Algebraic Doubleword Immediate X-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Registers</th>
<th>Destination Registers</th>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sradi RA,RS,SH</td>
<td>(Rc=0)</td>
<td>RA,RS,SH</td>
<td></td>
<td>64-bit fixed-point shift instruction with immediate shift right</td>
</tr>
<tr>
<td>sradi. RA,RS,SH</td>
<td>(Rc=1)</td>
<td>RA,RS,SH</td>
<td></td>
<td>64-bit fixed-point shift instruction with immediate sign extension</td>
</tr>
</tbody>
</table>

The contents of register `RS` are shifted right the number of bits specified by `(RB)57:63`. Bits shifted out of position 63 are lost. Bit 0 of `RS` is replicated to fill the vacated positions on the left. The result is placed into register `RA`. CA and CA32 are set to 1 if any 1-bits are shifted out of position 63; otherwise CA and CA32 are set to 0. A shift amount of zero causes RA to be set equal to |RS|, and CA and CA32 to be set to 0.

**Special Registers Altered:**

- `CA`  
- `CA32`  
- `CR0` (if `Rc=1`)
### Extend Sign Word and Shift Left Immediate

**XS-form**

| extswsli RA,RS,SH | (RC = 0) |
| extswsli RA,RS,SH | (RC = 1) |

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>sh</th>
<th>445</th>
<th>e</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{n} & \leftarrow s_{32} \mid s_{0:4} \\
\text{r} & \leftarrow \text{ROTL64} (\text{EXTS64}(R_{32:63}), n) \\
\text{m} & \leftarrow \text{MASK}(0, 63-n) \\
\text{RA} & \leftarrow r \& m
\end{align*}
\]

The contents of the low order 32 bits of RS are sign-extended to 64 bits and then shifted left SH bits. Bits shifted out of bit 0 are lost. Zeros are supplied to vacated bits on the right. The result is placed in register RA.

**Special Registers Altered:**

CR0 (if RC = 1)
### 3.3.15 Binary Coded Decimal (BCD) Assist Instructions

The Binary Coded Decimal Assist instructions operate on Binary Coded Decimal operands (cdtbcd and addg6s) and Decimal Floating-Point operands (cdt-bcd). See Chapter 5 for additional information.

#### Convert Declets To Binary Coded Decimal X-form

<table>
<thead>
<tr>
<th>cdtbcdc</th>
<th>RA, RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RS</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

- \( \text{do} \ i = 0 \ \text{to} \ 1 \)
- \( n \leftarrow i \times 32 \)
- \( R_{A+n+0:n+7} \leftarrow 0 \)
- \( R_{A+n+8:n+19} \leftarrow \text{DPD_TO_BCD}(R_{S+n+12:n+21}) \)
- \( R_{A+n+20:n+31} \leftarrow \text{DPD_TO_BCD}(R_{S+n+22:n+31}) \)

The low-order 20 bits of each word of register RS contain two declets which are converted to six, 4-bit BCD fields; each set of six, 4-bit BCD fields is placed into the low-order 24 bits of the corresponding word in RA. The high-order 8 bits in each word of RA are set to 0.

**Special Registers Altered:** None

#### Convert Binary Coded Decimal To Declets X-form

<table>
<thead>
<tr>
<th>cbcddc</th>
<th>RA, RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RS</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

- \( \text{do} \ i = 0 \ \text{to} \ 1 \)
- \( n \leftarrow i \times 32 \)
- \( R_{A+n+0:n+11} \leftarrow 0 \)
- \( R_{A+n+12:n+21} \leftarrow \text{BCD_TO_DPD}(R_{S+n+8:n+19}) \)
- \( R_{A+n+22:n+31} \leftarrow \text{BCD_TO_DPD}(R_{S+n+20:n+31}) \)

The low-order 24 bits of each word of register RS contain six, 4-bit BCD fields which are converted to two declets; each set of two declets is placed into the low-order 20 bits of the corresponding word in RA. The high-order 12 bits in each word of RA are set to 0.

If a 4-bit BCD field has a value greater than 9 the results are undefined.

**Special Registers Altered:** None
**Add and Generate Sixes XO-form**

```plaintext
addg6s RT,RA,RB
```

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>74</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
for i = 0 to 15
dci ← carry_out(RA4xi:63 + RB4xi:63)
c ← 4(dci) || 4(dci) || ... || 4(dci)
RT ← (~c) & 0x6666_6666_6666_6666
```

The contents of register RA are added to the contents of register RB. Sixteen carry bits are produced, one for each carry out of decimal position \( n \) (bit position \( 4\times n \)).

A doubleword is composed from the 16 carry bits, and placed into RT. The doubleword consists of a decimal six (0b0110) in every decimal digit position for which the corresponding carry bit is 0, and a zero (0b0000) in every position for which the corresponding carry bit is 1.

**Special Registers Altered:**

None

---

**Programming Note**

*addg6s* can be used to add or subtract two BCD operands. In these examples it is assumed that \( r0 \) contains 0x666...666. (BCD data formats are described in Section 5.3.)

Addition of the unsigned BCD operand in register RA to the unsigned BCD operand in register RB can be accomplished as follows.

```plaintext
add r1,RA,r0
add r2,r1,RB
addg6s RT,r1,RB
subf RT,RT,r2   # RT = RA + BCD RB
```

Subtraction of the unsigned BCD operand in register RA from the unsigned BCD operand in register RB can be accomplished as follows. (In this example it is assumed that RB is not register 0.)

```plaintext
addi r1,RB,1
nor r2,RA,RA   # one's complement of RA
add r3,r1,r2
addg6s RT,r1,r2
subf RT,RT,r3   # RT = RB - BCD RA
```

Additional instructions are needed to handle signed BCD operands, and BCD operands that occupy more than one register (e.g., unsigned BCD operands that have more than 16 decimal digits).
### 3.3.16 Byte-Reverse Instructions

**Byte-Reverse Halfword X-form**

\[ \text{brh} \quad \text{RA,RS} \]

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>RS</th>
<th>11</th>
<th>RA</th>
<th>16</th>
<th>219</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA [=] (RS)</td>
<td>8:35</td>
<td>| (RS)</td>
<td>0:7</td>
<td>| (RS)</td>
<td>24:31</td>
<td>| (RS)</td>
<td>16:23</td>
<td>| (RS)</td>
</tr>
</tbody>
</table>

The contents of bits 0:15 of register RS are placed into bits 0:15 of register RA in byte-reversed order.

The contents of bits 16:31 of register RS are placed into bits 16:31 of register RA in byte-reversed order.

The contents of bits 32:47 of register RS are placed into bits 32:47 of register RA in byte-reversed order.

The contents of bits 48:63 of register RS are placed into bits 48:63 of register RA in byte-reversed order.

**Special Registers Altered:**

None

**Byte-Reverse Word X-form**

\[ \text{brw} \quad \text{RA,RS} \]

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>RS</th>
<th>11</th>
<th>RA</th>
<th>16</th>
<th>155</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA [=] (RS)</td>
<td>24:31</td>
<td>| (RS)</td>
<td>16:23</td>
<td>| (RS)</td>
<td>8:35</td>
<td>| (RS)</td>
<td>0:7</td>
<td>| (RS)</td>
</tr>
</tbody>
</table>

The contents of bits 0:31 of register RS are placed into bits 0:31 of register RA in byte-reversed order.

The contents of bits 32:63 of register RS are placed into bits 32:63 of register RA in byte-reversed order.

**Special Registers Altered:**

None

**Byte-Reverse Doubleword X-form**

\[ \text{brd} \quad \text{RA,RS} \]

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>RS</th>
<th>11</th>
<th>RA</th>
<th>16</th>
<th>187</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA [=] (RS)</td>
<td>56:63</td>
<td>| (RS)</td>
<td>48:55</td>
<td>| (RS)</td>
<td>40:47</td>
<td>| (RS)</td>
<td>32:39</td>
<td>| (RS)</td>
</tr>
</tbody>
</table>

The contents of register RS are placed into register RA in byte-reversed order.

**Special Registers Altered:**

None
### 3.3.17 Move To/From Vector-Scalar Register Instructions

#### Move From VSR Doubleword X-form

**mfvsrd** RA,XS

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>S</th>
<th>RA</th>
<th>16</th>
<th>///</th>
<th>21</th>
<th>51</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>51</td>
<td>64</td>
</tr>
</tbody>
</table>

```plaintext
if SX=0 & MSR.FP=0 then FP_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()
```

GPR[RA] ← VSR[(32×SX+S)].dword[0]

Let XS be the value $32 \times SX + S$.

The contents of doubleword element 0 of VSR[XS] are placed into GPR[RA].

For SX=0, **mfvsrd** is treated as a Floating-Point instruction in terms of resource availability.

For SX=1, **mfvsrd** is treated as a Vector instruction in terms of resource availability.

**Extended Mnemonics:**

Extended mnemonics for Move From VSR Doubleword:

<table>
<thead>
<tr>
<th>Extended mnemonic:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfiprd RA,FRS</td>
<td>mfvsrd RA,FRS</td>
</tr>
<tr>
<td>mfvr d RA, VRS</td>
<td>mfvsrd RA, VRS+32</td>
</tr>
</tbody>
</table>

**Special Registers Altered:**

None

**Data Layout for mfvsrd**

```plaintext
src
tgt
```

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[XS].dword[0]</td>
<td>GPR[RA]</td>
</tr>
</tbody>
</table>

#### Move From VSR Lower Doubleword X-form

**mfvsrld** RA,XS

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>S</th>
<th>RA</th>
<th>16</th>
<th>///</th>
<th>21</th>
<th>307</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>307</td>
<td>64</td>
</tr>
</tbody>
</table>

```plaintext
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()
```

GPR[RA] ← VSR[(32×SX+S)].dword[1]

Let XS be the value $32 \times SX + S$.

The contents of doubleword 1 of VSR[XS] are placed into GPR[RA].

For SX=0, **mfvsrld** is treated as a VSX instruction in terms of resource availability.

For SX=1, **mfvsrld** is treated as a Vector instruction in terms of resource availability.

**Special Registers Altered:**

None

**Data Layout for mfvsrld**

```plaintext
src
tgt
```

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>VSR[XS].dword[1]</td>
</tr>
<tr>
<td>GPR[RA]</td>
<td></td>
</tr>
</tbody>
</table>
Move From VSR Word and Zero X-form

**mfvsrwz RA,XS**

\[
\begin{array}{cccccc}
31 & 6 & S & RA & /// & 115 \\
0 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[
\text{if SX=0 & MSR.FP=0 then FP_Unavailable()}
\]
\[
\text{if SX=1 & MSR.VEC=0 then Vector_Unavailable()}
\]

GPR[RA] ← EXTZ64(VSR[32×SX+S].word[1])

Let XS be the value 32×S + S.

The contents of word element 1 of VSR[XS] are placed into bits 32:63 of GPR[RA]. The contents of bits 0:31 of GPR[RA] are set to 0.

For SX=0, *mfvsrwz* is treated as a *Floating-Point* instruction in terms of resource availability.

For SX=1, *mfvsrwz* is treated as a *Vector* instruction in terms of resource availability.

**Extended Mnemonics:**

Extended mnemonics for Move To VSR Word and Zero:

**Extended mnemonic:**  
\text{mfprwz RA, FR5}  
\text{mfvrwz RA, VR5}

**Equivalent to:**  
\text{mfsvrwz RA, FR5}  
\text{mfsvrwz RA, VR5 + 32}

**Special Registers Altered**

None

**Data Layout for mfvsrwz**

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused VSR[XS].word[0]</td>
<td>unused GPR[RA]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Move To VSR Doubleword X-form

**mtvsrd XT,RA**

\[
\begin{array}{cccccc}
31 & 6 & T & RA & /// & 179 \\
0 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[
\text{if TX=0 & MSR.FP=0 then FP_Unavailable()}
\]
\[
\text{if TX=1 & MSR.VEC=0 then Vector_Unavailable()}
\]

VSR[32×TX+T].dword[0] ← GPR[RA]

VSR[32×TX+T].dword[1] ← 0xUUUU_UUUUUUUUUUU

Let XT be the value 32×TX + T.

The contents of GPR[RA] are placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, *mtvsrd* is treated as a *Floating-Point* instruction in terms of resource availability.

For TX=1, *mtvsrd* is treated as a *Vector* instruction in terms of resource availability.

**Extended Mnemonics:**

Extended mnemonics for Move To VSR Doubleword:

**Extended mnemonic:**  
\text{mftprd FRT, RA}  
\text{mtvrd VRT, RA}

**Equivalent to:**  
\text{mtsvrd FRT, RA}  
\text{mtsvrd VRT + 32, RA}

**Special Registers Altered**

None

**Data Layout for mtvsrd**

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR[RA]</td>
<td>VSR[XT].dword[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chapter 3. Fixed-Point Facility 121
Move To VSR Word Algebraic X-form

\[
\text{mtvsrwa} \quad \text{XT,RA}
\]

\[
\begin{array}{cccc|c}
0 & 31 & T & RA & 31 \\
6 & 11 & 16 & 21 & \text{/}/ \\
\end{array}
\]

if TX=0 & MSR.FP=0 then FP_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

VSR[32×TX+T].dword[0] ← EXT64(GPR[RA].bit[32:63])
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

Let XT be the value \(32 \times TX + T\).

The two's-complement integer in bits 32:63 of GPR[RA] is sign-extended to 64 bits and placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, \text{mtvsrwa} is treated as a Floating-Point instruction in terms of resource availability.

For TX=1, \text{mtvsrwa} is treated as a Vector instruction in terms of resource availability.

Extended Mnemonics:

Extended mnemonics for \text{Move To VSR Word Algebraic}:

\[
\begin{align*}
\text{Extended mnemonic:} & \quad \text{Equivalent to:} \\
\text{mtfprwa} & \quad \text{mtvsrwa} & \quad \text{FRT, RA} \\
\text{mtvra} & \quad \text{mtvsrwa} & \quad \text{VRT, RA}
\end{align*}
\]

Special Registers Altered
None

Data Layout for mtvsrwa

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{SRC} & \text{undefined} & \text{GPR[RA].bit[32:63]} & \text{tgt} \\
\hline
\text{VR[XT].dword[0]} & \text{undefined} & \text{VR[XT].dword[1]} & \text{0x0000_0000} & \text{VR[XT].word[0]} & \text{undefined} \\
\hline
\end{array}
\]

Move To VSR Word and Zero X-form

\[
\text{mtvsrwz} \quad \text{XT,RA}
\]

\[
\begin{array}{cccc|c}
0 & 31 & T & RA & 31 \\
6 & 11 & 16 & 21 & \text{/}/ \\
\end{array}
\]

if TX=0 & MSR.FP=0 then FP_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

VSR[32×TX+T].dword[0] ← EXTZ64(GPR[RA].word[1])
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

Let XT be the value \(32 \times TX + T\).

The contents of bits 32:63 of GPR[RA] are placed into word element 1 of VSR[XT]. The contents of word element 0 of VSR[XT] are set to 0.

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, \text{mtvsrwz} is treated as a Floating-Point instruction in terms of resource availability.

For TX=1, \text{mtvsrwz} is treated as a Vector instruction in terms of resource availability.

Extended Mnemonics:

Extended mnemonics for \text{Move To VSR Word and Zero}:

\[
\begin{align*}
\text{Extended mnemonic:} & \quad \text{Equivalent to:} \\
\text{mtfprwz} & \quad \text{mtvsrwz} & \quad \text{FRT, RA} \\
\text{mtvzwz} & \quad \text{mtvsrwz} & \quad \text{VRT, RA}
\end{align*}
\]

Special Registers Altered
None

Data Layout for mtvsrwz

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{SRC} & \text{undefined} & \text{GPR[RA].bit[32:63]} & \text{tgt} \\
\hline
\text{VR[XT].word[0]} & \text{undefined} & \text{VR[XT].word[1]} & \text{0x0000_0000} & \text{VR[VT].word[0]} & \text{undefined} \\
\hline
\end{array}
\]

Version 3.1
Power ISA™ I
**Move To VSR Double Doubleword X-form**

**mtvsrdd**  
XT,RA,RB

<table>
<thead>
<tr>
<th>X</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>435</th>
<th>7X</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if TX=0 & MSR.VSX=0 then VSX.Unavailable()  
if TX=1 & MSR.VEC=0 then Vector_Unavailable()  
if RA=0 then  
VSR[32×TX+T].dword[0] ← 0x0000_0000_0000_0000  
else  
VSR[32×TX+T].dword[0] ← GPR[RA]  
VSR[32×TX+T].dword[1] ← GPR[RB]

Let XT be the value 32×TX + T.

The contents of GPR[RA], or the value 0 if RA=0, are placed into doubleword 0 of VSR[XT].

The contents of GPR[RB] are placed into doubleword 1 of VSR[XT].

For TX=0, mtvsrdd is treated as a VSX instruction in terms of resource availability.

For TX=1, mtvsrdd is treated as a Vector instruction in terms of resource availability.

**Special Registers Altered:**  
None

**Data Layout for mtvsrdd**

|-------------|-------------|-------------|-------------|

**Move To VSR Word & Splat X-form**

**mtvsrws**  
XT,RA

<table>
<thead>
<tr>
<th>X</th>
<th>T</th>
<th>RA</th>
<th>///</th>
<th>403</th>
<th>7X</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if TX=0 & MSR.VSX=0 then VSX.Unavailable()  
if TX=1 & MSR.VEC=0 then Vector.Unavailable()  
VSR[32×TX+T].word[0] ← GPR[RA].bit[32:63]  

Let XT be the value 32×TX + T.

The contents of bits 32:63 of GPR[RA] are placed into each word element of VSR[XT].

For TX=0, mtvsrws is treated as a VSX instruction in terms of resource availability.

For TX=1, mtvsrws is treated as a Vector instruction in terms of resource availability.

**Special Registers Altered:**  
None

**Data Layout for mtvsrws**

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>GPR[RA].bit[32:63]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 32 64 96 127</td>
<td>0 32 64 96 127</td>
<td>0 32 64 96 127</td>
<td>0 32 64 96 127</td>
</tr>
</tbody>
</table>
3.3.18 Move To/From System Register Instructions

The *Move To Condition Register Fields* instruction has a preferred form; see Section 1.8.1, “Preferred Instruction Forms” on page 24. In the preferred form, the FXM field satisfies the following rule:

- Exactly one bit of the FXM field is set to 1.

**Extended mnemonics**

Extended mnemonics are provided for the *mtspr* and *mfspr* instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. An extended mnemonic is provided for the *mtcrf* instruction for compatibility with old software (written for a version of the architecture that preceded Version 2.00) that uses it to set the entire Condition Register. Some of these extended mnemonics are shown as examples with the relevant instructions. See Appendix C, “Assembler Extended Mnemonics” on page 1011 for additional extended mnemonics.

### Move To Special Purpose Register XFX-form

**mtspr** SPR,RS

<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>771</td>
<td>11000 00011</td>
<td>PMC1</td>
</tr>
<tr>
<td>772</td>
<td>11000 00100</td>
<td>PMC2</td>
</tr>
<tr>
<td>773</td>
<td>11000 00101</td>
<td>PMC3</td>
</tr>
<tr>
<td>774</td>
<td>11000 00110</td>
<td>PMC4</td>
</tr>
<tr>
<td>775</td>
<td>11000 01111</td>
<td>PMC5</td>
</tr>
<tr>
<td>776</td>
<td>11000 10000</td>
<td>PMC6</td>
</tr>
<tr>
<td>779</td>
<td>11000 01011</td>
<td>MMCR0</td>
</tr>
<tr>
<td>800</td>
<td>11001 00000</td>
<td>BESCR</td>
</tr>
<tr>
<td>801</td>
<td>11001 00001</td>
<td>BESCRSU</td>
</tr>
<tr>
<td>802</td>
<td>11001 00010</td>
<td>BESCRR</td>
</tr>
<tr>
<td>803</td>
<td>11001 00111</td>
<td>BESCRRU</td>
</tr>
<tr>
<td>804</td>
<td>11001 01000</td>
<td>EBBHR</td>
</tr>
<tr>
<td>805</td>
<td>11001 01011</td>
<td>EBBRU</td>
</tr>
<tr>
<td>806</td>
<td>11001 01110</td>
<td>BECRR</td>
</tr>
<tr>
<td>808</td>
<td>11001 01000</td>
<td>reserved&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>809</td>
<td>11001 01011</td>
<td>reserved&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>810</td>
<td>11001 01010</td>
<td>reserved&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>811</td>
<td>11001 01011</td>
<td>reserved&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>815</td>
<td>11001 01111</td>
<td>TAR&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>896</td>
<td>11100 00000</td>
<td>PPR</td>
</tr>
<tr>
<td>898</td>
<td>11100 00010</td>
<td>PPR32</td>
</tr>
</tbody>
</table>

1. Note that the order of the two 5-bit halves of the SPR number is reversed.
2. Accesses to these registers are no-ops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”. Otherwise, unless the SPR field contains 13 (denoting the AMR), the contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

The AMR (Authority Mask Register) is used for “storage protection.” This use, and operation of *mtspr* for the AMR, are described in Book III.

<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00000 000011</td>
<td>XER</td>
</tr>
<tr>
<td>3</td>
<td>00000 000111</td>
<td>DSCR</td>
</tr>
<tr>
<td>8</td>
<td>00000 010000</td>
<td>LR</td>
</tr>
<tr>
<td>9</td>
<td>00000 010001</td>
<td>CTR</td>
</tr>
<tr>
<td>13</td>
<td>00000 011001</td>
<td>AMR</td>
</tr>
<tr>
<td>256</td>
<td>01000 000000</td>
<td>VRSAVE</td>
</tr>
<tr>
<td>769</td>
<td>11000 000011</td>
<td>MMCR2</td>
</tr>
<tr>
<td>770</td>
<td>11000 000100</td>
<td>MMCR3</td>
</tr>
</tbody>
</table>

1. Note that the order of the two 5-bit halves of the SPR number is reversed.
2. Accesses to these registers are no-ops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”

If execution of this instruction is attempted specifying an SPR number that is not shown above, one of the following occurs:

- If spr<sub>4</sub> = 0, the illegal instruction error handler is invoked.
- If spr<sub>4</sub> = 1, the system privileged instruction error handler is invoked.

A complete description of this instruction can be found in Book III.

**Special Registers Altered:**

See above
Extended Mnemonics:

Examples of extended mnemonics for *Move To Special Purpose Register*:

<table>
<thead>
<tr>
<th>Extended mnemonic:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtxer Rx</td>
<td>mtspr 1,Rx</td>
</tr>
<tr>
<td>mtlr Rx</td>
<td>mtspr 8,Rx</td>
</tr>
<tr>
<td>mtctr Rx</td>
<td>mtspr 9,Rx</td>
</tr>
<tr>
<td>mtppr Rx</td>
<td>mtspr 896,Rx</td>
</tr>
<tr>
<td>mtppr32 Rx</td>
<td>mtspr 898,Rx</td>
</tr>
</tbody>
</table>

---

**Programming Note**

The AMR is part of the “context” of the program (see Book III). Therefore modification of the AMR requires “synchronization” by software. For this reason, most operating systems provide a system library program that application programs can use to modify the AMR.

---

**Compiler and Assembler Note**

For the *mtspr* and *mfspr* instructions, the SPR number coded in Assembler language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15.
Move From Special Purpose Register

XFX-form

\[ \text{mfspr} \text{ RT,SPR} \]

\[
\begin{array}{cccc}
0 & 31 & 6 & \text{RT} \\
1 & 31 & 11 & \text{spr} \\
2 & 31 & 21 & 339 \\
3 & 31 & 31 & 32 \\
\end{array}
\]

\[
h \leftarrow \text{spr}_{5:9} \ || \ \text{spr}_{0:4}
\]

\[
\text{switch} (n)
\]

\[
\begin{cases}
\text{case} (800, 809, 810, 811): & \text{default:} \\
\text{if} \ |\text{length}(\text{SPR}(n))| = 64 \text{ then} & \text{RT} \leftarrow \text{SPR}(n) \\
\text{else} & \text{RT} \leftarrow 32 \ || \ \text{SPR}(n)
\end{cases}
\]

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, "Reserved Fields, Reserved Values, and Reserved SPRs". Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

<table>
<thead>
<tr>
<th>decimal</th>
<th>spr&lt;sup&gt;T&lt;/sup&gt;</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00000 000001</td>
<td>XER</td>
</tr>
<tr>
<td>3</td>
<td>00000 00011</td>
<td>DSCR</td>
</tr>
<tr>
<td>8</td>
<td>00000 01000</td>
<td>LR</td>
</tr>
<tr>
<td>9</td>
<td>00000 01001</td>
<td>CTR</td>
</tr>
<tr>
<td>13</td>
<td>00000 01101</td>
<td>AMR</td>
</tr>
<tr>
<td>136</td>
<td>00100 01000</td>
<td>CTRL</td>
</tr>
<tr>
<td>256</td>
<td>01000 00000</td>
<td>VRSAVE</td>
</tr>
<tr>
<td>259</td>
<td>01000 00011</td>
<td>SPRG3</td>
</tr>
<tr>
<td>268</td>
<td>01000 01100</td>
<td>TB&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>269</td>
<td>01000 01101</td>
<td>TBU&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>768</td>
<td>11000 00000</td>
<td>SIER</td>
</tr>
<tr>
<td>769</td>
<td>11000 00001</td>
<td>MMCR2</td>
</tr>
<tr>
<td>770</td>
<td>11000 00010</td>
<td>MMCRA</td>
</tr>
<tr>
<td>771</td>
<td>11000 00011</td>
<td>PMC1</td>
</tr>
<tr>
<td>772</td>
<td>11000 00100</td>
<td>PMC2</td>
</tr>
<tr>
<td>773</td>
<td>11000 00101</td>
<td>PMC3</td>
</tr>
<tr>
<td>774</td>
<td>11000 00110</td>
<td>PMC4</td>
</tr>
<tr>
<td>775</td>
<td>11000 00111</td>
<td>PMC5</td>
</tr>
<tr>
<td>776</td>
<td>11000 01000</td>
<td>PMC6</td>
</tr>
<tr>
<td>779</td>
<td>11000 01011</td>
<td>MMCR0</td>
</tr>
<tr>
<td>780</td>
<td>11000 01100</td>
<td>SIAR</td>
</tr>
<tr>
<td>781</td>
<td>11000 01101</td>
<td>SDAR</td>
</tr>
<tr>
<td>782</td>
<td>11000 01110</td>
<td>MMCR1</td>
</tr>
<tr>
<td>800</td>
<td>11001 00000</td>
<td>BESCRS</td>
</tr>
</tbody>
</table>

1. Note that the order of the two 5-bit halves of the SPR number is reversed.
2. See Chapter 5 of Book II
3. Accesses to these SPRs are no-ops; see Section 1.3.3, "Reserved Fields, Reserved Values, and Reserved SPRs".

If execution of this instruction is attempted specifying an SPR number that is not shown above, one of the following occurs.

- If spr<sub>0</sub> = 0, the illegal instruction error handler is invoked.
- If spr<sub>0</sub> = 1, the system privileged instruction error handler is invoked.

A complete description of this instruction can be found in Book III.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for Move From Special Purpose Register:

Extended mnemonic: \( \text{mfxer Rx} \)
* Equivalent to: \( \text{mfspr Rx,1} \)

Extended mnemonic: \( \text{mflr Rx} \)
* Equivalent to: \( \text{mfspr Rx,8} \)

Extended mnemonic: \( \text{mtlr Rx} \)
* Equivalent to: \( \text{mfspr Rx,9} \)

Note

See the Notes that appear with \( \text{mtspr} \).
Move to CR from XER Extended X-form

```
mcrxrx  BF
```

<table>
<thead>
<tr>
<th>31</th>
<th>BF</th>
<th>//</th>
<th>//</th>
<th>//</th>
<th>576</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

CR4BF+32:4BF+35 ← XER01 0V32 CA C32

The contents of the OV, OV32, CA, and C32 are copied to Condition Register field BF.

Special Registers Altered:
CR field BF

Move To One Condition Register Field XFX-form

```
mtocrf  FXM,RS
```

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>1</th>
<th>FXM</th>
<th>//</th>
<th>144</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>12</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

count ← 0
do i = 0 to 7
  if FXMi = 1 then
    n ← i
    count ← count + 1
  if count = 1 then
    CR4n+32:4n+35 ← (RS)4n+32:4n+35
  else
    CR ← undefined

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field (0 ≤ n ≤ 7). The contents of bits 4×n+32:4×n+33 of register RS are placed into CR field n (CR bits 4×n+32:4×n+33). Otherwise, the contents of the Condition Register are undefined.

Special Registers Altered:
CR field selected by FXM

Move To Condition Register Fields XFX-form

```
mtrf  FXM,RS
```

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>0</th>
<th>FXM</th>
<th>//</th>
<th>144</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>12</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

mask ← 4(FXM0) || 4(FXM1) || ... || 4(FXM7)
CR ← (RS)32:63 & mask | (CR & ¬mask)

The contents of bits 32:63 of register RS are placed into the Condition Register under control of the field mask specified by FXM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If FXMi = 1 then CR field i (CR bits 4×i+32:4×i+33) is set to the contents of the corresponding field of the low-order 32 bits of RS.

Special Registers Altered:
CR fields selected by mask

Extended Mnemonics:
Example of extended mnemonics for Move To Condition Register Fields:

**Extended mnemonic:**

```
mect  Rx
```

**Equivalent to:**

```
mctr  0xFF,Rx
```
**Move From One Condition Register Field XFX-form**

```
<table>
<thead>
<tr>
<th>mfocrf</th>
<th>RT,FXM</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RT</td>
</tr>
<tr>
<td>0</td>
<td>FXM</td>
</tr>
</tbody>
</table>
```

RT ← undefined

```plaintext
count ← 0
do i = 0 to 7
   if FXMi = 1 then
      n ← i
   end if
   count ← count + 1
if count = 1 then
   RT ← 640
   RT4×n+32:4×n+35 ← CR4×n+32:4×n+35
```

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field \((0 \leq n \leq 7)\). The contents of CR field \(n\) (CR bits \(4×n+32:4×n+35\)) are placed into bits \(4×n+32:4×n+35\) of register RT, and the contents of the remaining bits of register RT are undefined. Otherwise, the contents of register RT are undefined.

If exactly one bit of the FXM field is set to 1, the contents of the remaining bits of register RT are set to 0's instead of being undefined as specified above.

**Special Registers Altered:**

None

**Programming Note**

Warning: `mfocrf` is not backward compatible with processors that comply with versions of the architecture that precede Version 2.08. Such processors may not set to 0 the bits of register RT that do not correspond to the specified CR field. If programs that depend on this clearing behavior are run on such processors, the programs may get incorrect results.

The POWER4, POWER5, POWER7 and POWER8 processors set to 0's all bytes of register RT other than the byte that contains the specified CR field. In the byte that contains the CR field, bits other than those containing the CR field may or may not be set to 0s.
### Set Boolean X-form

\[
\text{setb} \quad RT, BFA
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>BFA</th>
<th>14</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>128</td>
<td>1</td>
</tr>
</tbody>
</table>

- If \( CR_{BFA+32} = 1 \) then
  \[ RT \leftarrow 0xFFFF_FFFF_FFFF_FFFF \]
- Else if \( CR_{BFA+33} = 1 \) then
  \[ RT \leftarrow 0x0000_0000_0000_0001 \]
- Else
  \[ RT \leftarrow 0x0000_0000_0000_0000 \]

If the contents of bit 0 of CR field BFA are equal to 0b1, the contents of register RT are set to \( 0xFFFF_FFFF_FFFF_FFFF \).

Otherwise, if the contents of bit 1 of CR field BFA are equal to 0b1, the contents of register RT are set to \( 0x0000_0000_0000_0001 \).

Otherwise, the contents of register RT are set to \( 0x0000_0000_0000_0000 \).

**Special Registers Altered:** None

### Set Boolean Condition X-form

\[
\text{setbc} \quad RT, BI
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>BI</th>
<th>14</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>384</td>
<td>1</td>
</tr>
</tbody>
</table>

- \( RT = (CR_{BI}=1) ? 1 : 0 \)

If bit \( BI \) of the CR contains a 1, register RT is set to 1. Otherwise, register RT is set to 0.

**Special Registers Altered:** None

### Set Boolean Condition Reverse X-form

\[
\text{setbcr} \quad RT, BI
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>BI</th>
<th>14</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>416</td>
<td>1</td>
</tr>
</tbody>
</table>

- \( RT = (CR_{BI}=1) ? 0 : 1 \)

If bit \( BI \) of the CR contains a 1, register RT is set to 0. Otherwise, register RT is set to 1.

**Special Registers Altered:** None

### Set Negative Boolean Condition X-form

\[
\text{setnbc} \quad RT, BI
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>BI</th>
<th>14</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>448</td>
<td>1</td>
</tr>
</tbody>
</table>

- \( RT = (CR_{BI}=1) ? -1 : 0 \)

If bit \( BI \) of the CR contains a 1, register RT is set to -1. Otherwise, register RT is set to 0.

**Special Registers Altered:** None

### Set Negative Boolean Condition Reverse X-form

\[
\text{setnbcr} \quad RT, BI
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>BI</th>
<th>14</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>480</td>
<td>1</td>
</tr>
</tbody>
</table>

- \( RT = (CR_{BI}=1) ? 0 : -1 \)

If bit \( BI \) of the CR contains a 1, register RT is set to 0. Otherwise, register RT is set to -1.

**Special Registers Altered:** None
3.3.19 Prefixed No-Operation Instruction

Prefixed Nop MRR:“-form

pnop

Prefix:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>//</td>
<td>0</td>
</tr>
</tbody>
</table>

Suffix:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

1. Value must not correspond to a Branch instruction, an rfebb instruction, a context synchronizing instruction other than isync, or a "Service Processor Attention" instruction

No operation is performed.

Special Registers Altered:

None

Programming Note

The pnop instruction behaves as a b $+8 instruction regardless of its suffix. However, it does not cause any side effects such as modification of the Come From Address Register. (see Section 9.2 of Book III).

Programming Note

If the value in the suffix of a pnop instruction corresponds to a Branch instruction, an rfebb instruction, a context synchronizing instruction other than isync, or a "Service Processor Attention" instruction, the instruction form is invalid. The behavior associated with invalid form instructions is described in Section 1.8.2 on page 24. rfebb and isync are defined in Book II: Power ISA Virtual Environment Architecture. Context synchronization and other context synchronizing instructions are defined in Book III: Power ISA Operating Environment Architecture. Service Processor Attention is a reserved instruction; see Appendix C, "Reserved Instructions" on page 1347.)

This restriction eases hardware implementation complexity.

Engineering Note

Because the list of word instructions that must not be used as the suffix of pnop may change in the future, hardware should treat these invalid instruction forms of pnop either as a no-op or as an illegal instruction. This treatment enhances software compatibility. The choice may vary according to which of the word instructions is used as the suffix.
Chapter 4. Floating-Point Facility

4.1 Floating-Point Facility Overview

This chapter describes the registers and instructions that make up the Floating-Point Facility.

The processor (augmented by appropriate software support, where required) implements a floating-point system compliant with the ANSI/IEEE Standard 754-1985, “IEEE Standard for Binary Floating-Point Arithmetic” (hereafter referred to as “the IEEE standard”). That standard defines certain required “operations” (addition, subtraction, etc.). Herein, the term “floating-point operation” is used to refer to one of these required operations and to additional operations defined (e.g., those performed by Multiply-Add or Reciprocal Estimate instructions). A Non-IEEE mode is also provided. This mode, which may produce results not in strict compliance with the IEEE standard, allows shorter latency.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in floating-point registers; to move floating-point data between storage and these registers; and to manipulate the Floating-Point Status and Control Register explicitly.

These instructions are divided into two categories.

• computational instructions
  The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. They place status information into the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.6 through 4.6.8.

• non-computational instructions
  The non-computational instructions are those that perform loads and stores, move the contents of a floating-point register to another floating-point register possibly altering the sign, manipulate the Floating-Point Status and Control Register explicitly, and select the value from one of two floating-point registers based on the value in a third floating-point register. The operations performed by these instructions are not considered floating-point operations. With the exception of the instructions that manipulate the Floating-Point Status and Control Register explicitly, they do not alter the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.2 through 4.6.5, and 4.6.10.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number $2^{\text{exponent}}$. Encodings are provided in the data format to represent finite numeric values, $\pm\text{Infinity}$, and values that are “Not a Number” (NaN). Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. They may be used to indicate such things as uninitialized variables and can be produced by certain invalid operations.

There is one class of exceptional events that occur during instruction execution that is unique to the Floating-Point Facility: the Floating-Point Exception. Floating-point exceptions are signaled with bits set in the Floating-Point Status and Control Register (FPSCR). They can cause the system floating-point enabled exception error handler to be invoked, precisely or imprecisely, if the proper control bits are set.

Floating-Point Exceptions

The following floating-point exceptions are detected by the processor:

• Invalid Operation Exception (VX)
  SNaN (VXSNaN)
  Infinity–Infinity (VXSI)
  Infinity–Infinity (VXDI)
  Zero–Zero (VXZDZ)
  Infinity–Zero (VXIMZ)
  Invalid Compare (VXVC)
  Software-Defined Condition (VXSOFT)
  Invalid Square Root (VXSQRT)


Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. See Section 4.2.2, “Floating-Point Status and Control Register” on page 132 for a description of these exception and enable bits, and Section 4.4, “Floating-Point Exceptions” on page 140 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.

4.2 Floating-Point Facility Registers

4.2.1 Floating-Point Registers

Implementations of this architecture provide 32 floating-point registers (FPRs). The floating-point instruction formats provide 5-bit fields for specifying the FPRs to be used in the execution of the instruction. The FPRs are numbered 0-31. See Figure 45 on page 132.

Each FPR contains 64 bits that support the floating-point double format. Every instruction that interprets the contents of an FPR as a floating-point value uses the floating-point double format for this interpretation.

The computational instructions, and the Move and Select instructions, operate on data located in FPRs and, with the exception of the Compare instructions, place the result value into an FPR and optionally (when \( Rc=1 \)) place status information into the Condition Register.

Load Double and Store Double instructions are provided that transfer 64 bits of data between storage and the FPRs with no conversion. Load Single instructions are provided to transfer and convert floating-point values in floating-point single format from storage to the same value in floating-point double format in the FPRs. Store Single instructions are provided to transfer and convert floating-point values in floating-point double format from the FPRs to the same value in floating-point single format in storage.

Instructions are provided that manipulate the Floating-Point Status and Control Register and the Condition Register explicitly. Some of these instructions copy data from an FPR to the Floating-Point Status and Control Register or vice versa.

The computational instructions and the Select instruction accept values from the FPRs in double format. For single-precision arithmetic instructions, all input values must be representable in single format; if they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if \( Rc=1 \)), are undefined.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Reserved</td>
</tr>
<tr>
<td>29:31</td>
<td>Decimal Rounding Mode (( DRN ))</td>
</tr>
<tr>
<td></td>
<td>See Section 5.2.1, “DFP Usage of Floating-Point Registers” on page 188.</td>
</tr>
<tr>
<td>32</td>
<td>Floating-Point Exception Summary (( FX ))</td>
</tr>
<tr>
<td></td>
<td>Every floating-point instruction, except ( mtfsf ) and ( mtfsf ), implicitly sets ( FX ) to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. ( mcrfs ), ( mtfsi ), ( mtfsi ), ( mtfsb0 ), and ( mtfsb1 ) can alter ( FX ) explicitly.</td>
</tr>
</tbody>
</table>
Floating-Point Enabled Exception Summary (FX)
This bit is the OR of all the floating-point exception bits masked by their respective enable bits. mcrfs, mtsfsfi, mtsfsf, mtfsb0, and mtfsb1 cannot alter FX explicitly.

Floating-Point Invalid Operation Exception Summary (XX)
This bit is the OR of all the Invalid Operation exception bits. mcrfs, mtsfsfi, mtsfsf, mtfsb0, and mtfsb1 cannot alter XX explicitly.

Floating-Point Overflow Exception (OX)
See Section 4.4.3, “Overflow Exception” on page 143.

Floating-Point Underflow Exception (UX)
See Section 4.4.4, “Underflow Exception” on page 144.

Floating-Point Zero Divide Exception (ZX)
See Section 4.4.2, “Zero Divide Exception” on page 142.

Floating-Point Inexact Exception (XX)
See Section 4.4.5, “Inexact Exception” on page 144.

XX is a sticky version of FI (see below). Thus the following rules completely describe how XX is set by a given instruction.

- If the instruction affects FI, the new value of XX is obtained by ORing the old value of XX with the new value of FI.
- If the instruction does not affect FI, the value of XX is unchanged.

Floating-Point Invalid Operation Exception (SNaN) (VXSNAN)
See Section 4.4.1, “Invalid Operation Exception” on page 142.

Floating-Point Invalid Operation Exception (∞ - ∞) (VX1S1)
See Section 4.4.1.

Floating-Point Invalid Operation Exception (∞ + ∞) (VX1DI)
See Section 4.4.1.

Floating-Point Invalid Operation Exception (0 ÷ 0) (VX2D2)
See Section 4.4.1.

Floating-Point Invalid Operation Exception (∞ ÷ 0) (VX1MZ)
See Section 4.4.1.

Floating-Point Invalid Operation Exception (Invalid Compare) (VXVC)
See Section 4.4.1.

Floating-Point Fraction Rounded (FR)
The last Arithmetic or Rounding and Conversion instruction incremented the fraction during rounding. See Section 4.3.6, “Rounding” on page 139. This bit is not sticky.

Floating-Point Fraction Inexact (FI)
The last Arithmetic or Rounding and Conversion instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 4.3.6. This bit is not sticky.

See the definition of XX, above, regarding the relationship between FI and XX.

Floating-Point Result Flags (FPRF)
Arithmetic, rounding, and Convert From Integer instructions set this field based on the result placed into the target register and on the target precision, except that if any portion of the result is undefined then the value placed into FPRF is undefined. Floating-point Compare instructions set this field based on the relative values of the operands being compared. For Convert To Integer instructions, the value placed into FPRF is undefined. Additional details are given below.

A single-precision operation that produces a denormalized result sets FPRF to indicate a denormalized number. When possible, single-precision denormalized numbers are represented in normalized double format in the target register.

Floating-Point Result Class Descriptor (C)
Arithmetic, rounding, and Convert From Integer instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 47 on page 135.
Floating-Point Condition Code (FPCC)
Floating-point Compare instructions set one of the FPCC bits to 1 and the other three FPCC bits to 0. Arithmetic, rounding, and Convert From Integer instructions may set the FPCC bits with the C bit, to indicate the class of the result as shown in Figure 47 on page 135. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

Floating-Point Less Than or Negative (FL or <)
Floating-Point Greater Than or Positive (FG or >)
Floating-Point Equal or Zero (FE or =)
Floating-Point Unordered or NaN (FU or ?)
Reserved
Floating-Point Invalid Operation Exception (Software-Defined Condition) (VXSOFT)
This bit can be altered only by mcrfs, mtfsfi, mtfsf, mtfsb0, or mtfsb1. See Section 4.4.1.

Programming Note
FPSCR\textsubscript{VXSOFT} can be used by software to indicate the occurrence of an arbitrary, software-defined, condition that is to be treated as an Invalid Operation Exception. For example, the bit could be set by a program that computes a base 10 logarithm if the supplied input is negative.

Floating-Point Invalid Operation Exception (Invalid Square Root) (VXSQRT)
See Section 4.4.1.
Floating-Point Invalid Operation Exception (Invalid Integer Convert) (VXCVI)
See Section 4.4.1.
Floating-Point Invalid Operation Exception Enable (VE)
See Section 4.4.1.
Floating-Point Overflow Exception Enable (OE)
See Section 4.4.3, “Overflow Exception” on page 143.
Floating-Point Underflow Exception Enable (UE)
See Section 4.4.4, “Underflow Exception” on page 144.
Floating-Point Zero Divide Exception Enable (ZE)
See Section 4.4.2, “Zero Divide Exception” on page 142.

Floating-Point Inexact Exception Enable (XE)
See Section 4.4.5, “Inexact Exception” on page 144.
Floating-Point Non-IEEE Mode (NI)
Floating-point non-IEEE mode is optional. If floating-point non-IEEE mode is not implemented, this bit is treated as reserved, and the remainder of the definition of this bit does not apply.

If floating-point non-IEEE mode is implemented, this bit has the following meaning.
0 The processor is not in floating-point non-IEEE mode (i.e., all floating-point operations conform to the IEEE standard).
1 The processor is in floating-point non-IEEE mode.

When the processor is in floating-point non-IEEE mode, the remaining FPSCR bits may have meanings different from those given in this document, and floating-point operations need not conform to the IEEE standard. The effects of executing a given floating-point instruction with FPSCR\textsubscript{NI}=1, and any additional requirements for using non-IEEE mode, are implementation-dependent. The results of executing a given instruction in non-IEEE mode may vary between implementations, and between different executions on the same implementation.

Programming Note
When the processor is in floating-point non-IEEE mode, the results of floating-point operations may be approximate, and performance for these operations may be better, more predictable, or less data-dependent than when the processor is not in non-IEEE mode. For example, in non-IEEE mode an implementation may return 0 instead of a denormalized number, and may return a large number instead of an infinity.

Floating-Point Rounding Control (RN)
See Section 4.3.6, “Rounding” on page 139.
00 Round to nearest
01 Round toward zero
10 Round toward +Infinity
11 Round toward -Infinity
4.3 Floating-Point Data

4.3.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed-length formats. The format may be a 32-bit single format for a single-precision value or a 64-bit double format for a double-precision value. The single format may be used for data in storage. The double format may be used for data in storage and for data in floating-point registers.

The lengths of the exponent and the fraction fields differ between these two formats. The structure of the single and double formats is shown below.

![Figure 48. Floating-point single format](image)

![Figure 49. Floating-point double format](image)

Values in floating-point format are composed of three fields:

- **S**: sign bit
- **EXP**: exponent+bias
- **FRACTION**: fraction

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized numbers and is located in the unit bit position (i.e., the first bit to the left of the binary point). Values representable within the two floating-point formats can be specified by the parameters listed in Figure 50.

![Figure 50. IEEE floating-point fields](image)

The architecture requires that the FPRs of the Floating-Point Facility support the floating-point double format only.

4.3.2 Value Representation

This architecture defines numeric and non-numeric values representable within each of the two supported formats. The numeric values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The non-numeric values representable are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible however to define restricted operations among numbers and infinities as defined below. The relative location on the real number line for each of the defined entities is shown in Figure 51.

![Figure 51. Approximation to real numbers](image)

The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

**Binary floating-point numbers**

Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.
**Normalized numbers** (± NOR)
These are values that have a biased exponent value in the range:

- 1 to 254 in single format
- 1 to 2046 in double format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

\[ \text{NOR} = (-1)^s \times 2^E \times (1.\text{fraction}) \]

where \( s \) is the sign, \( E \) is the unbiased exponent, and \( 1.\text{fraction} \) is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

The ranges covered by the magnitude (\( M \)) of a normalized floating-point number are approximately equal to:

- **Single Format:**
  \[ 1.2 \times 10^{-38} \leq M \leq 3.4 \times 10^{38} \]
- **Double Format:**
  \[ 2.2 \times 10^{-308} \leq M \leq 1.8 \times 10^{308} \]

**Zero values** (± 0)
These are values that have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (i.e., comparison regards +0 as equal to –0).

**Denormalized numbers** (± 0)
These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as follows:

\[ \text{DEN} = (-1)^s \times 2^{E_{\text{min}}} \times (0.\text{fraction}) \]

where \( E_{\text{min}} \) is the minimum representable exponent value (−126 for single-precision, −1022 for double-precision).

**Infinities** (± ∞)
These are values that have the maximum biased exponent value:

- 255 in single format
- 2047 in double format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

\(-∞ < \text{every finite number} < +∞\)

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 4.4.1, “Invalid Operation Exception” on page 142.

For comparison operations, +Infinity compares equal to +Infinity and -Infinity compares equal to -Infinity.

**Not a Numbers** (NaNs)
These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (i.e., NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0 then the NaN is a Signaling NaN; otherwise it is a Quiet NaN.

Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions. Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation Exception is disabled (\( \text{VE} = 0 \)). Quiet NaNs propagate through all floating-point operations except ordered comparison, Floating Round to Single-Precision, and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a disabled Invalid Operation Exception, then the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

If \( \text{FRA} \) is a NaN
then \( \text{FRT} \leftarrow \text{FRA} \)
else if \( \text{FRB} \) is a NaN
then if instruction is \( \text{frsp} \)
    then \( \text{FRT} \leftarrow \text{FRB} \)
    else \( \text{FRT} \leftarrow \text{FRB} \)
else if \( \text{FRC} \) is a NaN
then \( \text{FRT} \leftarrow \text{FRC} \)
else if generated QNaN
then \( \text{FRT} \leftarrow \text{generated QNaN} \)

If the operand specified by \( \text{FRA} \) is a NaN, then that NaN is stored as the result. Otherwise, if the operand specified by \( \text{FRB} \) is a NaN (if the instruction specifies an \( \text{FRB} \) operand), then that NaN is stored as the result, with the low-order 29 bits of the result set to 0 if the instruction is \( \text{frsp} \). Otherwise, if the operand specified by \( \text{FRC} \) is a NaN (if the instruction specifies an \( \text{FRC} \) operand), then that NaN is stored as the result. Otherwise, if a QNaN was generated due to a disabled Invalid Operation Exception, then that QNaN is stored as the result. If a QNaN is to be generated as a result, then the QNaN generated has a sign bit of 0, an exponent field of all 1s, and a high-order fraction bit of 1 with all other fraction bits 0. Any instruction that generates a QNaN as
the result of a disabled Invalid Operation Exception generates this QNaN (i.e., \texttt{0x7FF8_0000_0000_0000}).

A double-precision NaN is considered to be representable in single format if and only if the low-order 29 bits of the double-precision NaN's fraction are zero.

### 4.3.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same sign, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation \( x - y \) is the same as the sign of the result of the add operation \( x + (-y) \).

When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except Round toward \( -\infty \), in which mode the sign is negative.

- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.

- The sign of the result of a Square Root or Reciprocal Square Root Estimate operation is always positive, except that the square root of \(-0\) is \(-0\) and the reciprocal square root of \(-0\) is \(-\infty\).

- The sign of the result of a Round to Single-Precision, or Convert From Integer, or Round to Integer operation is the sign of the operand being converted.

For the Multiply-Add instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

### 4.3.4 Normalization and Denormalization

The intermediate result of an arithmetic or \texttt{frsp} instruction may require normalization and/or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or rounding instruction produces an intermediate result which carries out of the significand, or in which the significand is nonzero but has a leading zero bit, it is not a normalized number and must be normalized before it is stored. For the carry-out case, the significand is shifted right one bit, with a one shifted into the leading significand bit, and the exponent is incremented by one. For the leading-zero case, the significand is shifted left while decremented its exponent by one for each bit shifted, until the leading significant bit becomes one. The Guard bit and the Round bit (see Section 4.5.1, “Execution Model for IEEE Operations” on page 145) participate in the shift with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result may have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be “Tiny” and the stored result is determined by the rules described in Section 4.4.4, “Underflow Exception”. These rules may require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format’s minimum value. If any significant bits are lost in this shifting process then “Loss of Accuracy” has occurred (See Section 4.4.4, “Underflow Exception” on page 144) and Underflow Exception is signaled.

### 4.3.5 Data Handling and Precision

Most of the Floating-Point Facility Architecture, including all computational, Move, and Select instructions, use the floating-point double format to represent data in the FPRs. Single-precision and integer-valued operands may be manipulated using double-precision operations. Instructions are provided to coerce these values from a double format operand. Instructions are also provided for manipulations which do not require double-precision. In addition, instructions are provided to access a true single-precision representation in storage, and a fixed-point integer representation in GPRs.

#### 4.3.5.1 Single-Precision Operands

For single format data, a format conversion from single to double is performed when loading from storage into an FPR and a format conversion from double to single is performed when storing from an FPR to storage. No floating-point exceptions are caused by these instructions. An instruction is provided to explicitly convert a double format operand in an FPR to single-precision. Floating-point single-precision is enabled with four types of instruction.

1. Load Floating-Point Single

   This form of instruction accesses a single-precision operand in single format in storage, converts it to double format, and loads it into an FPR. No floating-point exceptions are caused by these instructions.
2. Round to Floating-Point Single-Precision

The Floating Round to Single-Precision instruction rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into an FPR in double format. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of the Floating Round to Single-Precision instruction, this operation does not alter the value.

3. Single-Precision Arithmetic Instructions

This form of instruction takes operands from the FPRs in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single format. Status bits, in the FPSCR and optionally in the Condition Register, are set to reflect the single-precision result. The result is then converted to double format and placed into an FPR. The result lies in the range supported by the single format.

If any input value is not representable in single format and either OE=1 or UE=1, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1), are undefined.

For fres[] or frsqrtes[], if the input value is finite and has an unbiased exponent greater than +127, the input value is interpreted as an Infinity.

4. Store Floating-Point Single

This form of instruction converts a double-precision operand to single format and stores that operand into storage. No floating-point exceptions are caused by these instructions. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding three types.)

When the result of a Load Floating-Point Single, Floating Round to Single-Precision, or single-precision arithmetic instruction is stored in an FPR, the low-order 29 FRACTION bits are zero.

--- Programming Note ---

The Floating Round to Single-Precision instruction is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions and by fcfid) to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by a Floating Round to Single-Precision instruction.

--- Programming Note ---

A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value is representable in single format.

Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, single-precision data and instructions should be used.

4.3.5.2 Integer-Valued Operands

Instructions are provided to round floating-point operands to integer values in floating-point format. To facilitate exchange of data between the floating-point and fixed-Point facilities, instructions are provided to convert between floating-point double format and fixed-point integer format in an FPR. Computation on integer-valued operands may be performed using arithmetic instructions of the required precision. (The results may not be integer values.) The two groups of instructions provided specifically to support integer-valued operands are described below.

1. Floating Round to Integer

The Floating Round to Integer instructions round a double-precision operand to an integer value in floating-point double format. These instructions may cause Invalid Operation (VXSNAN) exceptions. See Sections 4.3.6 and 4.5.1 for more information about rounding.

2. Floating Convert To/From Integer

The Floating Convert To/Integer instructions convert a double-precision operand to a 32-bit or 64-bit signed fixed-point integer format. Variants are provided both to perform rounding based on
the value of FPSCR\textsubscript{RN} and to round toward zero. These instructions may cause Invalid Operation (\texttt{VXSNAN, VXCVI}) and Inexact exceptions. The Floating Convert From Integer instruction converts a 64-bit signed fixed-point integer to a double-precision floating-point integer. Because of the limitations of the source format, only an Inexact exception may be generated.

### 4.3.6 Rounding

The material in this section applies to operations that have numeric operands (i.e., operands that are not infinities or NaNs). Rounding the intermediate result of such an operation may cause an Overflow Exception, an Underflow Exception, or an Inexact Exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 4.3.2, “Value Representation” and Section 4.4, “Floating-Point Exceptions” for the cases not covered here.

The Arithmetic and Rounding and Conversion instructions round their intermediate results. With the exception of the Estimate instructions, these instructions produce an intermediate result that can be regarded as having infinite precision and unbounded exponent range. All but two groups of these instructions normalize or denormalize the intermediate result prior to rounding and then place the final result into the target FPR in double format. The Floating Round to Integer and Floating Convert To Integer instructions with biased exponents ranging from 1022 through 1074 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 1075. (Intermediate results with biased exponents 1075 or larger are already integers, and with biased exponents 1021 or less round to zero.) After rounding, the final result for Floating Round to Integer is normalized and put in double format, and for Floating Convert To Integer is converted to a signed fixed-point integer.

FPSCR bits \texttt{FR} and \texttt{FI} generally indicate the results of rounding. Each of the instructions which rounds its intermediate result sets these bits. If the fraction is incremented during rounding then \texttt{FR} is set to 1, otherwise \texttt{FR} is set to 0. If the result is inexact then \texttt{FI} is set to 1, otherwise \texttt{FI} is set to zero. The Round to Integer instructions are exceptions to this rule, setting \texttt{FR} and \texttt{FI} to 0. The Estimate instructions set \texttt{FR} and \texttt{FI} to undefined values. The remaining floating-point instructions do not alter \texttt{FR} and \texttt{FI}.

Four user-selectable rounding modes are provided through the Floating-Point Rounding Control field in the FPSCR. See Section 4.2.2, “Floating-Point Status and Control Register”. These are encoded as follows.

<table>
<thead>
<tr>
<th>RN</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Round to Nearest</td>
</tr>
<tr>
<td>01</td>
<td>Round toward Zero</td>
</tr>
<tr>
<td>10</td>
<td>Round toward +Infinity</td>
</tr>
<tr>
<td>11</td>
<td>Round toward -Infinity</td>
</tr>
</tbody>
</table>

Let \( Z \) be the intermediate arithmetic result or the operand of a convert operation. If \( Z \) can be represented exactly in the target format, then the result in all rounding modes is \( Z \) as represented in the target format. If \( Z \) cannot be represented exactly in the target format, let \( Z_1 \) and \( Z_2 \) bound \( Z \) as the next larger and next smaller numbers representable in the target format. Then \( Z_1 \) or \( Z_2 \) can be used to approximate the result in the target format.

Figure 52 shows the relation of \( Z, Z_1, \text{ and } Z_2 \) in this case. The following rules specify the rounding in the four modes. “LSB” means “least significant bit”.

![Figure 52. Selection of Z1 and Z2](image-url)
4.4 Floating-Point Exceptions

This architecture defines the following floating-point exceptions:

- **Invalid Operation Exception**
  - SNaN
  - Infinity–Infinity
  - Infinity–Zero
  - Zero–Zero
  - Invalid Compare
  - Software-Defined Condition
  - Invalid Square Root
  - Invalid Integer Convert
- **Zero Divide Exception**
- **Overflow Exception**
- **Underflow Exception**
- **Inexact Exception**

These exceptions, other than Invalid Operation Exception due to Software-Defined Condition, may occur during execution of computational instructions. An Invalid Operation Exception due to Software-Defined Condition occurs when a Move To FPSCR instruction sets VXSOFT to 1.

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 141), whether and how the system floating-point enabled exception error handler is invoked. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

A single instruction, other than `mtfsfi` or `mtsf`, may set more than one exception bit only in the following cases:

- **Inexact Exception** may be set with **Overflow Exception**.
- **Inexact Exception** may be set with **Underflow Exception**.
- **Invalid Operation Exception** (SNaN) is set with **Invalid Operation Exception** (`-∞×0`) for **Multiply-Add** instructions for which the values being multiplied are infinity and zero and the value being added is an SNaN.
- **Invalid Operation Exception** (SNaN) may be set with **Invalid Operation Exception** (Invalid Compare) for **Compare Ordered** instructions.
- **Invalid Operation Exception** (SNaN) may be set with **Invalid Operation Exception** (Invalid Integer Convert) for **Convert To Integer** instructions.

When an exception occurs the writing of a result to the target register may be suppressed or a result may be delivered, depending on the exception.

The writing of a result to the target register is suppressed for the following kinds of exception, so that there is no possibility that one of the operands is lost:

- **Enabled Invalid Operation**
- **Enabled Zero Divide**

For the remaining kinds of exception, a result is generated and written to the destination specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exception that deliver a result are the following:

- **Disabled Invalid Operation**
- **Disabled Zero Divide**
- **Disabled Overflow**
- **Disabled Underflow**
- **Disabled Inexact**
- **Enabled Overflow**
- **Enabled Underflow**
- **Enabled Inexact**

Subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of “traps” and “trap handlers”. In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the “trap enabled” case; the expectation is that the exception will be detected by software, which will revise the result. An FPSCR exception enable bit of 0 causes generation of the “default result” value specified for the “trap disabled” (or “no trap occurs” or “trap is not implemented”) case; the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to 0 and Ignore Exceptions Mode (see below) should be used. In this case the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur: software can inspect the FPSCR exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1 and a mode other than Ignore Exceptions Mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled float-
ing-point exception occurs. The system floating-point enabled exception error handler is also invoked if a Move To FPSCR instruction causes an exception bit and the corresponding enable bit both to be 1; the Move To FPSCR instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The location of these bits and the requirements for altering them are described in Book III. (The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception.) The effects of the four possible settings of these bits are as follows.

<table>
<thead>
<tr>
<th>FE0</th>
<th>FE1</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Ignore Exceptions Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Imprecise Nonrecoverable Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The system floating-point enabled exception error handler is invoked at some point after the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction may have been used by or may have affected subsequent instructions that are executed before the error handler is invoked.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Imprecise Recoverable Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler that it can identify the excepting instruction and the operands, and correct the result. No results produced by the excepting instruction have been used by or have affected subsequent instructions that are executed before the error handler is invoked.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Precise Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.</td>
</tr>
</tbody>
</table>

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. The instruction at which the system floating-point enabled exception error handler is invoked has completed if it is the excepting instruction and there is only one such instruction. Otherwise it has not begun execution (or may have been partially executed in some cases, as described in Book III).

**Programming Note**

In any of the three non-Precise modes, a Floating-Point Status and Control Register instruction can be used to force any exceptions, due to instructions initiated before the Floating-Point Status and Control Register instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In either of the Imprecise modes, a Floating-Point Status and Control Register instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to instructions initiated before the Floating-Point Status and Control Register instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

The last sentence of the paragraph preceding this Programming Note can apply only in the Imprecise modes, or if the mode has just been changed from Ignore Exceptions Mode to some other mode. (It always applies in the latter case.)

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to 0.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.
4.4.1 Invalid Operation Exception

4.4.1.1 Definition
An Invalid Operation Exception occurs when an operand is invalid for the specified operation. The invalid operations are:
- Any floating-point operation on a Signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities ($\infty - \infty$)
- Division of infinity by infinity ($\infty / \infty$)
- Division of zero by zero (0/0)
- Multiplication of infinity by zero ($\infty \times 0$)
- Ordered comparison involving a NaN (Invalid Compare)
- Square root or reciprocal square root of a negative number (Invalid Square Root)
- Integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN (Invalid Integer Convert)

An Invalid Operation Exception also occurs when an mtfsfi, mtfsf, or mtfsb1 instruction is executed that sets VXSOFT to 1 (Software-Defined Condition).

4.4.1.2 Action
The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

When Invalid Operation Exception is enabled ($VE=1$) and an Invalid Operation Exception occurs, the following actions are taken:
1. One or two Invalid Operation Exceptions are set
   - VXSNAN (if SNaN)
   - VXSI (if any NaN)
   - VXDI (if $\infty - \infty$)
   - VXZDZ (if 0/0)
   - VXNZ (if $\infty \times 0$)
   - VXVC (if invalid compare)
   - VXSOFT (if software-defined condition)
   - VXSV (if invalid square root)
   - VXCV (if invalid integer convert)

2. If the operation is an arithmetic or Floating Round to Single-Precision operation, the target FPR is set to a Quiet NaN.
   - FR FI are set to zero
   - FPRF is set to indicate the class of the result (Quiet NaN)

3. If the operation is a convert to 64-bit integer operation, the target FPR is set as follows:
   - FRT is set to the most positive 64-bit integer if the operand in FRB is a positive number or $+\infty$,
   - FRT is set to the most negative 64-bit integer if the operand in FRB is a negative number, $-\infty$, or NaN.
   - FR FI are set to zero
   - FPRF is undefined

4. If the operation is a compare, FR FI C are unchanged

When Invalid Operation Exception is disabled ($VE=0$) and an Invalid Operation Exception occurs, the following actions are taken:
1. One or two Invalid Operation Exceptions are set
   - VXSNAN (if SNaN)
   - VXSI (if any NaN)
   - VXDI (if $\infty - \infty$)
   - VXZDZ (if 0/0)
   - VXNZ (if $\infty \times 0$)
   - VXVC (if invalid compare)
   - VXSOFT (if software-defined condition)
   - VXSV (if invalid square root)
   - VXCV (if invalid integer convert)

2. If the operation is an arithmetic or Floating Round to Integer operation, the target FPR is set as follows:
   - FRT is set to the most positive 64-bit integer if the operand in FRB is a positive number or $+\infty$,
   - FRT is set to the most negative 64-bit integer if the operand in FRB is a negative number, $-\infty$, or NaN.
   - FR FI are set to zero
   - FPRF is undefined

3. If the operation is a convert to 32-bit integer operation, the target FPR is set as follows:
   - FRT0:31 ← undefined
   - FRT32:63 are set to the most positive 32-bit integer if the operand in FRB is a positive number or $+\infty$,
   - FRT32:63 are set to the most negative 32-bit integer if the operand in FRB is a negative number, $-\infty$, or NaN.
   - FR FI are set to zero
   - FPRF is undefined

4. If the operation is a compare, FR FI C are unchanged

5. If the operation is a compare, FR FI C are changed

6. If an mtfsfi, mtfsf, or mtfsb1 instruction is executed that sets VXSOFT to 1, the FPSCR is set as specified in the instruction description.

4.4.2 Zero Divide Exception

4.4.2.1 Definition
A Zero Divide Exception occurs when a Divide instruction is executed with a zero divisor value and a finite nonzero dividend value. It also occurs when a Reciprocal Estimate instruction (fre[s] or frsqrte[s]) is executed with an operand value of zero.
4.4.2.2 Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled (ZE=1) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   \[ ZX \leftarrow 1 \]
2. The target FPR is unchanged
3. \( FR, FI \) are set to zero
4. \( FPRF \) is unchanged

When Zero Divide Exception is disabled (ZE=0) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   \[ ZX \leftarrow 1 \]
2. The target FPR is set to \( \pm \)Infinity, where the sign is determined by the XOR of the signs of the operands
3. \( FR, FI \) are set to zero
4. \( FPRF \) is set to indicate the class and sign of the result (\( \pm \)Infinity)

4.4.3 Overflow Exception

4.4.3.1 Definition

An Overflow Exception occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

4.4.3.2 Action

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled (OE=1) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   \[ OX \leftarrow 1 \]
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536
3. For single-precision arithmetic instructions and the Floating Round to Single-Precision instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192
4. The adjusted rounded result is placed into the target FPR
5. \( FR \) is undefined
6. \( FI \) is set to 1
7. \( FPRF \) is set to indicate the class and sign of the result (\( \pm \)Infinity or \( \pm \)Normal Number)

When Overflow Exception is disabled (OE=0) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   \[ OX \leftarrow 1 \]
2. Inexact Exception is set
   \[ XX \leftarrow 1 \]
3. The result is determined by the rounding mode (RN) and the sign of the intermediate result as follows:
   - Round to Nearest
     Store \( \pm \)Infinity, where the sign is the sign of the intermediate result
   - Round toward Zero
     Store the format’s largest finite number with the sign of the intermediate result
   - Round toward +Infinity
     For negative overflow, store the format’s most negative finite number; for positive overflow, store +Infinity
   - Round toward −Infinity
     For negative overflow, store −Infinity; for positive overflow, store the format’s largest finite number
4. The result is placed into the target FPR
5. \( FR \) is undefined
6. \( FI \) is set to 1
7. \( FPRF \) is set to indicate the class and sign of the result (\( \pm \)Infinity or \( \pm \)Normal Number)
4.4.4 Underflow Exception

4.4.4.1 Definition

Underflow Exception is defined separately for the enabled and disabled states:

- **Enabled:**
  Underflow occurs when the intermediate result is “Tiny”.

- **Disabled:**
  Underflow occurs when the intermediate result is “Tiny” and there is “Loss of Accuracy”.

A “Tiny” result is detected before rounding, when a non-zero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is “Tiny” and Underflow Exception is disabled (\(UE=0\)) then the intermediate result is denormalized (see Section 4.3.4, “Normalization and Denormalization” on page 137) and rounded (see Section 4.3.6, “Rounding” on page 139) before being placed into the target FPR.

“Loss of Accuracy” is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

4.4.4.2 Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled (\(UE=1\)) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   \[UX \leftarrow 1\]
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536
3. For single-precision arithmetic instructions and the Floating Round to Single-Precision instruction, the exponent of the normalized intermediate result is adjusted by adding 192
4. The adjusted rounded result is placed into the target FPR
5. \(FPRF\) is set to indicate the class and sign of the result (± Normalized Number)

When Underflow Exception is disabled (\(UE=0\)) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   \[UX \leftarrow 1\]
2. The rounded result is placed into the target FPR
3. \(FPRF\) is set to indicate the class and sign of the result (± Normalized Number, ± Denormalized Number, or ± Zero)

4.4.5 Inexact Exception

4.4.5.1 Definition

An Inexact Exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled Overflow Exception or an enabled Underflow Exception, an Inexact Exception also occurs only if the significands of the rounded result and the intermediate result differ.)
2. The rounded result overflows and Overflow Exception is disabled.

4.4.5.2 Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When an Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set
   \[XX \leftarrow 1\]
2. The rounded or overflowed result is placed into the target FPR
3. \(FPRF\) is set to indicate the class and sign of the result

Programming Note

The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an Underflow Exception, to simulate a “trap disabled” environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.
4.5 Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (i.e., operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 4.3.2 and Section 4.4 for the cases not covered here.

Although the double format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

• Underflow during multiplication using a denormalized operand.
• Overflow during division using a denormalized divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision floating-point operations to have either (or both) single-precision or double-precision operands, but states that single-precision floating-point operations should not accept double-precision operands. The Power ISA follows these guidelines; double-precision arithmetic instructions can have operands of either or both precisions, while single-precision arithmetic instructions require all operands to be single-precision. Double-precision arithmetic instructions and \texttt{fcfidd} produce double-precision values, while single-precision arithmetic instructions produce single-precision values.

For arithmetic instructions, conversions from double-precision to single-precision must be done explicitly by software, while conversions from single-precision to double-precision are done implicitly.

4.5.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the \texttt{FRACTION} is a 23-bit field, and the single-precision Guard, Round, and Sticky bits (described in this section) are logically adjacent to the 23-bit \texttt{FRACTION} field.

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:55 comprise the significand of the intermediate result.

\[
\begin{array}{cccc}
S & C & L & \text{FRACTION} & G & R & X \\
0 & 1 & & 53 & 54 & 55 \\
\end{array}
\]

Figure 53. IEEE 64-bit execution model

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

The \texttt{FRACTION} is a 52-bit field that accepts the fraction of the operand.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for postnormalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the \texttt{R} bit, due either to shifting the accumulator right or to other generation of low-order result bits. The G and \texttt{R} bits participate in the left shifts with zeros being shifted into the \texttt{R} bit. Figure 54 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number next lower in magnitude (NL), and the representable number next higher in magnitude (NH).

\[
\begin{array}{cccc}
G & R & X & \text{Interpretation} \\
0 & 0 & 0 & \text{IR is exact} \\
0 & 0 & 1 & \text{IR closer to NL} \\
0 & 1 & 0 & \text{IR midway between NL and NH} \\
0 & 1 & 1 & \\
1 & 0 & 0 & \text{IR closer to NH} \\
1 & 0 & 1 & \\
1 & 1 & 0 & \\
1 & 1 & 1 & \\
\end{array}
\]

Figure 54. Interpretation of G, R, and X bits

For arithmetic instructions, conversions from double-precision to single-precision must be done explicitly by software, while conversions from single-precision to double-precision are done implicitly.
The significand of the intermediate result is prepared for rounding by shifting its contents right, if required, until the least significant bit to be retained is in the low-order bit position of the fraction. Four user-selectable rounding modes are provided through RN as described in Section 4.3.6, “Rounding” on page 139. Using Z1 and Z2 as defined on page 139, the rules for rounding in each mode are as follows.

- **Round to Nearest**
  
  **Guard bit = 0**
  The result is truncated. (Result exact (GRX=000) or closest to next lower value in magnitude (GRX=001, 010, or 011))

  **Guard bit = 1**
  Depends on Round and Sticky bits:

  **Case a**
  If the Round or Sticky bit is 1 (inclusive), the result is incremented. (Result closest to next higher value in magnitude (GRX=101, 110, or 111))

  **Case b**
  If the Round and Sticky bits are 0 (result midway between closest representable values), then if the low-order bit of the result is 1 the result is incremented. Otherwise (the low-order bit of the result is 0) the result is truncated (this is the case of a tie rounded to even).

- **Round toward Zero**
  Choose the smaller in magnitude of Z1 or Z2. If the Guard, Round, or Sticky bit is nonzero, the result is inexact.

- **Round toward + Infinity**
  Choose Z1.

- **Round toward - Infinity**
  Choose Z2.

If rounding results in a carry into C, the significand is shifted right one position and the exponent is incremented by one. This yields an inexact result, and possibly also exponent overflow. If any of the Guard, Round, or Sticky bits is nonzero, then the result is also inexact. Fraction bits are stored to the target FPR. For Floating Round to Integer, Floating Round to Single-Precision, and single-precision arithmetic instructions, low-order zeros must be appended as appropriate to fill out the double-precision fraction.
4.5.2 Execution Model for Multiply-Add Type Instructions

The Power ISA provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar except that the \texttt{FRACTION} field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:106 comprise the significand of the intermediate result.

\begin{figure}[h]
\centering
\begin{tabular}{cccc}
S & C & L & FRACTION \hline
0 & 1 & 2 & 3 & \multicolumn{1}{c}{106}
\end{tabular}
\caption{Multiply-add 64-bit execution model}
\end{figure}

The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), then the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the \texttt{FRACTION} and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the \texttt{FRACTION}) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount that is added to that exponent to make it equal to the other input’s exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X’ bit. The add operation also produces a result conforming to the above model with the X’ bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the X’ bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 57 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Format} & \textbf{Guard} & \textbf{Round} & \textbf{Sticky} \\
\hline
Double & 53 & 54 & OR of 55:105, X’ \\
Single & 24 & 25 & OR of 26:105, X’ \\
\hline
\end{tabular}
\caption{Location of the Guard, Round, and Sticky bits in the multiply-add execution model}
\end{figure}

The rules for rounding the intermediate result are the same as those given in Section 4.5.1.

If the instruction is \textit{Floating Negative Multiply-Add} or \textit{Floating Negative Multiply-Subtract}, the final result is negated.
4.6 Floating-Point Facility Instructions

4.6.1 Floating-Point Storage Access Instructions

The Storage Access instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.10.3, “Effective Address Calculation” on page 29.

Programming Note

The la extended mnemonic permits computing an effective address as a Load or Store instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at that address. This extended mnemonic is described in Section C.10, “Miscellaneous Mnenomics” on page 1022.

4.6.1.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (Store only), or if the program attempts to access storage that is unavailable.
4.6.2 Floating-Point Load Instructions

There are three basic forms of load instruction: single-precision, double-precision, and integer. The integer form is provided by the Load Floating-Point as Integer Word Algebraic instruction, described on page 153. Because the FPRs support only floating-point double format, single-precision Load Floating-Point instructions convert single-precision data to double format prior to loading the operand into the target FPR. The conversion and loading steps are as follows.

Let \( \text{WORD}_{0..31} \) be the floating-point single-precision operand accessed from storage.

**Normalized Operand**

\[
\text{if } \text{WORD}_{0..8} > 0 \text{ and } \text{WORD}_{0..8} < 255 \text{ then}
\]
\[
\begin{align*}
\text{FRT}_0 : 1 & \leftarrow \text{WORD}_0 : 1 \\
\text{FRT}_2 & \leftarrow \neg \text{WORD}_0 \\
\text{FRT}_3 & \leftarrow \neg \text{WORD}_0 \\
\text{FRT}_4 & \leftarrow \neg \text{WORD}_0 \\
\text{FRT}_5 : 63 & \leftarrow \text{WORD}_2 : 31 \| \text{290}
\end{align*}
\]

**Denormalized Operand**

\[
\text{if } \text{WORD}_{0..8} = 0 \text{ and } \text{WORD}_{0..31} \neq 0 \text{ then}
\]
\[
\begin{align*}
\text{sign} & \leftarrow \text{WORD}_0 \\
\text{exp} & \leftarrow -126 \\
\text{frac}_0 : 52 & \leftarrow 0b0 \| \text{WORD}_0 : 31 \| \text{290} \\
\text{normalize the operand} \\
\text{do while } \text{frac}_0 = 0 \\
\text{frac}_0 : 52 & \leftarrow \text{frac}_1 : 52 \| 0b0 \\
\text{exp} & \leftarrow \text{exp} - 1 \\
\text{FRT}_0 & \leftarrow \text{sign} \\
\text{FRT}_1 : 11 & \leftarrow \text{exp} + 1023 \\
\text{FRT}_2 : 63 & \leftarrow \text{frac}_1 : 52
\end{align*}
\]

**Zero / Infinity / NaN**

\[
\text{if } \text{WORD}_{0..8} = 255 \text{ or } \text{WORD}_{0..31} = 0 \text{ then}
\]
\[
\begin{align*}
\text{FRT}_0 : 1 & \leftarrow \text{WORD}_0 : 1 \\
\text{FRT}_2 & \leftarrow \text{WORD}_0 \\
\text{FRT}_3 & \leftarrow \text{WORD}_0 \\
\text{FRT}_4 & \leftarrow \text{WORD}_0 \\
\text{FRT}_5 : 63 & \leftarrow \text{WORD}_0 : 31 \| \text{290}
\end{align*}
\]

For double-precision Load Floating-Point instructions and for the Load Floating-Point as Integer Word Algebraic instruction no conversion is required, as the data from storage are copied directly into the FPR.

Many of the Load Floating-Point instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA≠0, the effective address is placed into register RA and the storage element (word or doubleword) addressed by EA is loaded into FRT.

**Note:** Recall that RA and RB denote General Purpose Registers, while FRT denotes a Floating-Point Register.
Load Floating-Point Single D-form

\[ \text{lfs FRT,D(RA)} \]

<table>
<thead>
<tr>
<th>48</th>
<th>FRT</th>
<th>RA</th>
<th>D</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>----</td>
</tr>
</tbody>
</table>

Prefixed Load Floating-Point Single

\[ \text{MLS:D-form} \]

\[ \text{plfs FRT,D(RA),R} \]

Prefix:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>d0</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>12</td>
<td>31</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th>48</th>
<th>FRT</th>
<th>RA</th>
<th>D</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>----</td>
</tr>
</tbody>
</table>

if "lfs" then
\[ \text{EA} \leftarrow (\text{RA}|0) + \text{EXTS64(D)} \]
if "plfs" & R=0 then
\[ \text{EA} \leftarrow (\text{RA}|0) + \text{EXTS64}(d0|d1) \]
if "plfs" & R=1 then
\[ \text{EA} \leftarrow (\text{CIA} + \text{EXTS64}(d0|d1)) \]
\[ \text{FRT} \leftarrow \text{DOUBLE(MEM(EA, 4))} \]

For \text{lfs}, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value D, sign-extended to 64 bits.

For \text{plfs} with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0|d1, sign-extended to 64 bits.

For \text{plfs} with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0|d1, sign-extended to 64 bits.

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 149) and placed into register FRT.

**Special Registers Altered:**
None

Extended mnemonics for Prefixes Load Floating-Point Single:

Extended mnemonic: \[ \text{plfs Fx,value(Ry)} \] Equivalent to: \[ \text{plfs Fx,value(Ry),0} \] \[ \text{plfs Fx,value} \] \[ \text{plfs Fx,value(0),1} \]

Load Floating-Point Single Indexed X-form

\[ \text{lfsx FRT,RA,RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>FRT</th>
<th>RA</th>
<th>RB</th>
<th>535</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if RA = 0 then \[ b \leftarrow 0 \] else \[ b \leftarrow (\text{RA}) \] \[ \text{EA} \leftarrow b + (\text{RB}) \] \[ \text{FRT} \leftarrow \text{DOUBLE(MEM(EA, 4))} \]

Let the effective address (EA) be the sum (RA|0) + (RB).

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 149) and placed into register FRT.

**Special Registers Altered:**
None

Load Floating-Point Single with Update

\[ \text{lfsu FRT,D(RA)} \]

\[ \text{EA} \leftarrow (\text{RA}) + \text{EXTS}(\text{D}) \] \[ \text{FRT} \leftarrow \text{DOUBLE(MEM(EA, 4))} \] \[ \text{RA} \leftarrow \text{EA} \]

Let the effective address (EA) be the sum (RA) + D.

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 149) and placed into register FRT.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**
None
**Load Floating-Point Single with Update Indexed X-form**

`lfsux FRT,RA,RB`

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>567</th>
<th>31</th>
</tr>
</thead>
</table>

Let the effective address (EA) be the sum (RA) + (RB).

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 149) and placed into register FRT.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**
None
Load Floating-Point Double D-form

Ifd  FRT,D(RA)

<table>
<thead>
<tr>
<th>50</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>D</th>
<th>31</th>
</tr>
</thead>
</table>

Prefixed Load Floating-Point Double
MLS:D-form

Plfd  FRT,D(RA),R

Prefix:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>0 //</th>
<th>0 //</th>
<th>d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>14</td>
<td>31</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th>50</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>d1</th>
<th>31</th>
</tr>
</thead>
</table>

if "ifd" then
EA ← (RA|0) + EXTS64(D)
if "plfd" & R=0 then
EA ← (RA|0) + EXTS64(d0||d1)
if "plfd" & R=1 then
EA ← CIA + EXTS64(d0||d1)

FRT ← MEM(EA, 8)

For ifd, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value D, sign-extended to 64 bits.

For plfd with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

For plfd with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0||d1, sign-extended to 64 bits.

The doubleword in storage addressed by EA is loaded into register FRT.

For plfd, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

Special Registers Altered:
None

Extended mnemonics for Prefixed Load Floating-Point Double:

Extended mnemonic:           Equivalent to:
plfd  fx,value(Ry)           plfd  fx,value(Ry),0
plfd  fx,value               plfd  fx,value(0),1

Load Floating-Point Double Indexed X-form

Ifdx  FRT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>D</th>
<th>599</th>
<th>1</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
FRT ← MEM(EA, 8)

Let the effective address (EA) be the sum (RA|0) + (RB).

The doubleword in storage addressed by EA is loaded into register FRT.

Special Registers Altered:
None

Load Floating-Point Double with Update D-form

Ifdu  FRT,D(RA)

<table>
<thead>
<tr>
<th>51</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>D</th>
<th>631</th>
<th>1</th>
</tr>
</thead>
</table>

EA ← (RA) + EXTS(D)
FRT ← MEM(EA, 8)
RA ← EA

Let the effective address (EA) be the sum (RA) + D.

The doubleword in storage addressed by EA is loaded into register FRT.

EA is placed into register RA.
If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Load Floating-Point Double with Update Indexed X-form

Ifdux  FRT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>631</th>
<th>1</th>
</tr>
</thead>
</table>

EA ← (RA) + (RB)
FRT ← MEM(EA, 8)
RA ← EA

Let the effective address (EA) be the sum (RA) + (RB).

The doubleword in storage addressed by EA is loaded into register FRT.

EA is placed into register RA.
If RA=0, the instruction form is invalid.

Special Registers Altered:
None
**Load Floating-Point as Integer Word Algebraic Indexed X-form**

Ifiwxax \( \text{FRT,RA,RB} \)

<table>
<thead>
<tr>
<th>31</th>
<th>FRT</th>
<th>RA</th>
<th>RB</th>
<th>855</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( \text{FRT} \leftarrow \text{EXT5(MEM(EA, 4))} \)

Let the effective address \( (EA) \) be the sum \( (RA|0) + (RB) \).

The word in storage addressed by \( EA \) is loaded into \( \text{FRT}_{32:63} \). \( \text{FRT}_{0:31} \) are filled with a copy of bit 0 of the loaded word.

**Special Registers Altered:**

None

---

**Load Floating-Point as Integer Word & Zero Indexed X-form**

Ifiwxzx \( \text{FRT,RA,RB} \)

<table>
<thead>
<tr>
<th>31</th>
<th>FRT</th>
<th>RA</th>
<th>RB</th>
<th>887</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( \text{FRT} \leftarrow 0 | | \text{MEM(EA, 4)} \)

Let the effective address \( (EA) \) be the sum \( (RA|0) + (RB) \).

The word in storage addressed by \( EA \) is loaded into \( \text{FRT}_{32:63} \). \( \text{FRT}_{0:31} \) are set to 0.

**Special Registers Altered:**

None
4.6.3 Floating-Point Store Instructions

There are three basic forms of store instruction: single-precision, double-precision, and integer. The integer form is provided by the Store Floating-Point as Integer Word instruction, described on page 158. Because the FPRs support only floating-point double format for floating-point data, single-precision Store Floating-Point instructions convert double-precision data to single format prior to storing the operand into storage. The conversion steps are as follows.

Let $\text{WORD}_{0:31}$ be the word in storage written to.

**No Denormalization Required (includes Zero / Infinity / NaN)**

\[
\text{if } \text{FRS}_1:13 > \text{BBB} \text{ or } \text{FRS}_1:63 = 0 \text{ then}
\]
\[
\text{WORD}_{0:1} \leftarrow \text{FRS}_0:1
\]
\[
\text{WORD}_{2:31} \leftarrow \text{FRS}_5:34
\]

**Denormalization Required**

\[
\text{if } 874 \leq \text{FRS}_1:11 \leq \text{BBB} \text{ then}
\]
\[
\text{sign} \leftarrow \text{FRS}_0
\]
\[
\text{exp} \leftarrow \text{FRS}_1:11 - 1023
\]
\[
\text{frac}_{0:52} \leftarrow 0b1 || \text{FRS}_{0:63}
\]
\[
\text{denormalize operand}
\]
\[
\text{do while } \text{exp} < -126
\]
\[
\text{frac}_{0:51} \leftarrow 0b0 || \text{frac}_{0:51}
\]
\[
\text{exp} \leftarrow \text{exp} + 1
\]
\[
\text{WORD}_{0} \leftarrow \text{sign}
\]
\[
\text{WORD}_{2:8} \leftarrow 0x00
\]
\[
\text{WORD}_{9:31} \leftarrow \text{frac}_{1:23}
\]
\[
\text{else } \text{WORD} \leftarrow \text{undefined}
\]

Notice that if the value to be stored by a single-precision Store Floating-Point instruction is larger in magnitude than the maximum number representable in single format, the first case above (No Denormalization Required) applies. The result stored in WORD is then a well-defined value, but is not numerically equal to the value in the source register (i.e., the result of a single-precision Load Floating-Point from WORD will not compare equal to the contents of the original source register).

For double-precision Store Floating-Point instructions and for the Store Floating-Point as Integer Word instruction no conversion is required, as the data from the FPR are copied directly into storage.

Many of the Store Floating-Point instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA≠0, the effective address is placed into register RA.

**Note:** Recall that RA and RB denote General Purpose Registers, while FRS denotes a Floating-Point Register.
Store Floating-Point Single D-form

\[
\begin{array}{cccc}
\text{stfs} & \text{FRS, D(RA)} \\
52 & 6 & 11 & 16 & D \\
0 & & & & 31
\end{array}
\]

**Prefixed Store Floating-Point Single MLS:D-form**

\[
\begin{array}{cccc}
\text{pstfs} & \text{FRS, D(RA), R} \\
52 & 6 & 11 & 16 & d1 \\
0 & & & & 31
\end{array}
\]

\[
\text{if } \text{"stfs" then} \\
\text{EA} \leftarrow (\text{RA}|0) + \text{EXTS}(D) \\
\text{if } \text{"pstfs" & R=0 then} \\
\text{EA} \leftarrow (\text{RA}|0) + \text{EXTS}(d0|d1) \\
\text{if } \text{"pstfs" & R=1 then} \\
\text{EA} \leftarrow \text{CIA} + \text{EXTS}(d0|d1) \\
\text{MEM(EA, 4)} \leftarrow \text{SINGLE}(\text{FRS})
\]

For **stfs**, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value D, sign-extended to 64 bits.

For **pstfs** with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0|d1, sign-extended to 64 bits.

For **pstfs** with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0|d1, sign-extended to 64 bits.

The contents of register FRS are converted to single format (see page 154) and stored into the word in storage addressed by EA.

**Special Registers Altered:** None

**Extended mnemonics for Prefixed Store Floating-Point Single:**

- \text{pstfs} Fx, value(Ry)
- \text{pstfs} Fx, value

**Store Floating-Point Single Indexed X-form**

\[
\begin{array}{cccc}
\text{stfsx} & \text{FRS, RA, RB} \\
53 & 6 & 11 & 16 & D \\
0 & & & & 31
\end{array}
\]

\[
\text{if } \text{RA} = 0 \text{ then } b \leftarrow 0 \\
\text{else} \\
\text{b} \leftarrow (\text{RA}) \\
\text{EA} \leftarrow b + |\text{RB}| \\
\text{MEM(EA, 4)} \leftarrow \text{SINGLE}(\text{FRS})
\]

Let the effective address (EA) be the sum (RA|0) + |RB|.

The contents of register FRS are converted to single format (see page 154) and stored into the word in storage addressed by EA.

**Special Registers Altered:** None

**Store Floating-Point Single with Update D-form**

\[
\begin{array}{cccc}
\text{stfsu} & \text{FRS, D(RA)} \\
53 & 6 & 11 & 16 & D \\
0 & & & & 31
\end{array}
\]

\[
\text{EA} \leftarrow (\text{RA}) + \text{EXTS}(D) \\
\text{MEM(EA, 4)} \leftarrow \text{SINGLE}(\text{FRS}) \\
\text{RA} \leftarrow \text{EA}
\]

Let the effective address (EA) be the sum (RA) + D.

The contents of register FRS are converted to single format (see page 154) and stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:** None
**Store Floating-Point Single with Update Indexed X-form**

stfsux FRS,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>30-24</th>
<th>23-16</th>
<th>15-8</th>
<th>7-0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FRS</td>
<td>RA</td>
<td>RB</td>
<td>695</td>
</tr>
</tbody>
</table>

\[
EA \leftarrow (RA) + (RB) \\
MEM(EA, 4) \leftarrow \text{SINGLE}((FRS)) \\
RA \leftarrow EA
\]

Let the effective address \(EA\) be the sum \((RA) + (RB)\).

The contents of register \(FRS\) are converted to single format (see page 154) and stored into the word in storage addressed by \(EA\).

\(EA\) is placed into register \(RA\).

If \(RA=0\), the instruction form is invalid.

**Special Registers Altered:**
None
**Store Floating-Point Double D-form**

<table>
<thead>
<tr>
<th>stfd</th>
<th>FRS, D(RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>FRS</td>
</tr>
<tr>
<td>6</td>
<td>RA</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

**Prefixed Store Floating-Point Double MLS:D-form**

<table>
<thead>
<tr>
<th>pstfd</th>
<th>FRS, D(RA), R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Prefix:**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>0</th>
<th>R</th>
<th>d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>0</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>54</td>
<td>FRS</td>
<td>RA</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>11</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Suffix:**

If “stfd” then

\[ EA \leftarrow (RA|0) + EXTS64(D) \]

If “pstfd” & R=0 then

\[ EA \leftarrow (RA|0) + EXTS64(d0||d1) \]

If “pstfd” & R=1 then

\[ EA \leftarrow CIA + EXTS64(d0||d1) \]

The contents of register FRS are stored into the double-word in storage addressed by EA.

**Special Registers Altered:**

None

**Extended mnemonics for Prefix:***

- **Prefixed:**
  - stfd: Fx, value(Ry)
  - pstfd: Fx, value(Ry)

**Equivalent to:**

- **Prefixed:**
  - pstfd: Fx, value(Ry)
  - stfd: Fx, value(Ry)

**Store Floating-Point Double Indexed X-form**

<table>
<thead>
<tr>
<th>stfdx</th>
<th>FRS, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>FRS</td>
</tr>
<tr>
<td>6</td>
<td>RA</td>
</tr>
<tr>
<td>11</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>D</td>
</tr>
<tr>
<td>727</td>
<td></td>
</tr>
</tbody>
</table>

Let the effective address (EA) be the sum \((RA|0) + (RB)\).

The contents of register FRS are stored into the double-word in storage addressed by EA.

**Special Registers Altered:**

None

**Store Floating-Point Double with Update D-form**

<table>
<thead>
<tr>
<th>stfdu</th>
<th>FRS, D(RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>FRS</td>
</tr>
<tr>
<td>6</td>
<td>RA</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

\[ EA \leftarrow (RA|0) + EXTS(D) \]

MEM(EA, B) \leftarrow (FRS)

RA \leftarrow EA

Let the effective address (EA) be the sum \((RA|0) + D\).

The contents of register FRS are stored into the double-word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**

None
Store Floating-Point Double with Update Indexed X-form

`stfdux FRS,RA,RB`

<table>
<thead>
<tr>
<th>31</th>
<th>FRS</th>
<th>RA</th>
<th>RB</th>
<th>759</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>759</td>
</tr>
</tbody>
</table>

EA ← (RA) + (RB)
MEM|EA, 8| ← (FRS)
RA ← EA

Let the effective address (EA) be the sum (RA) + (RB).

The contents of register FRS are stored into the double-word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point as Integer Word Indexed X-form

`stfiwx FRS,RA,RB`

<table>
<thead>
<tr>
<th>31</th>
<th>FRS</th>
<th>RA</th>
<th>RB</th>
<th>983</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>983</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← |RA|
EA ← b + (RB)
MEM|EA, 4| ← (FRS)32:63

Let the effective address (EA) be the sum (RA|0)+(RB).

(FRS)32:63 are stored, without conversion, into the word in storage addressed by EA.

If the contents of register FRS were produced, either directly or indirectly, by a Load Floating-Point Single instruction, a single-precision Arithmetic instruction, or frsp, then the value stored is undefined. (The contents of register FRS are produced directly by such an instruction if FRS is the target register for the instruction. The contents of register FRS are produced indirectly by such an instruction if FRS is the final target register of a sequence of one or more Floating-Point Move instructions, with the input to the sequence having been produced directly by such an instruction.)

Special Registers Altered:
None
4.6.4 Floating-Point Load and Store Double Pair Instructions [Phased-Out]

For \texttt{lfdp[x]}, the doubleword-pair in storage addressed by \( EA \) is loaded into an even-odd pair of FPRs with the even-numbered FPR being loaded with the leftmost doubleword from storage and the odd-numbered FPR being loaded with the rightmost doubleword.

For \texttt{stfdp[x]}, the content of an even-odd pair of FPRs is stored into the doubleword-pair in storage addressed by \( EA \), with the even-numbered FPR being stored into the leftmost doubleword in storage and the odd-numbered FPR being stored into the rightmost doubleword.

<table>
<thead>
<tr>
<th>Load Floating-Point Double Pair DS-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{lfdp}</td>
</tr>
<tr>
<td>( S7 )</td>
</tr>
<tr>
<td>57</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{if RA} &= 0 \text{ then } b \leftarrow 0 \\
\text{else} &\quad b \leftarrow (RA) \\
\text{EA} &\leftarrow b + \text{EXTS(DS||0b00)} \\
\text{FRTpeven} &\leftarrow \text{MEM(EA, 8)} \\
\text{FRTpodd} &\leftarrow \text{MEM(EA+8, 8)}
\end{align*}
\]

Let the effective address (\( EA \)) be the sum \( (RA|0) + (DS||0b00) \).

The doubleword in storage addressed by \( EA \) is placed into the even-numbered register of \( \text{FRTp} \).

The doubleword in storage addressed by \( EA+8 \) is placed into the odd-numbered register of \( \text{FRTp} \).

If \( \text{FRTp} \) is odd, the instruction form is invalid.

**Special Registers Altered:** None

<table>
<thead>
<tr>
<th>Load Floating-Point Double Pair Indexed X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{lfdpx}</td>
</tr>
<tr>
<td>( S7 )</td>
</tr>
<tr>
<td>57</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{if RA} &= 0 \text{ then } b \leftarrow 0 \\
\text{else} &\quad b \leftarrow (RA) \\
\text{EA} &\leftarrow b + (RB) \\
\text{FRTpeven} &\leftarrow \text{MEM(EA, 8)} \\
\text{FRTpodd} &\leftarrow \text{MEM(EA+8, 8)}
\end{align*}
\]

Let the effective address (\( EA \)) be the sum \( (RA|0) + (RB) \).

The doubleword in storage addressed by \( EA \) is placed into the even-numbered register of \( \text{FRTp} \).

The doubleword in storage addressed by \( EA+8 \) is placed into the odd-numbered register of \( \text{FRTp} \).

If \( \text{FRTp} \) is odd, the instruction form is invalid.

**Special Registers Altered:** None

---

**Programming Note**

The instructions described in this section should not be used to access an operand in DFP Extended format when the processor is in Little-Endian mode.
Store Floating-Point Double Pair DS-form

\[ \text{stfdp} \quad \text{FRSp,DS(RA)} \]

<table>
<thead>
<tr>
<th>61</th>
<th>FRSp</th>
<th>RA</th>
<th>DS</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>30 31</td>
</tr>
</tbody>
</table>

If RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow |\text{RA}| \)
\( \text{EA} \leftarrow b + \text{EXTS(DS)||0b00|} \)
\( \text{MEM(EA, 8)} \leftarrow \text{FRSp}_{\text{even}} \)
\( \text{MEM(EA+8, 8)} \leftarrow \text{FRSp}_{\text{odd}} \)

Let the effective address (EA) be the sum \( (\text{RA}|0) + (\text{DS||0b00}) \).

The contents of the even-numbered register of FRSp are stored into the doubleword in storage addressed by EA.
The contents of the odd-numbered register of FRSp are stored into the doubleword in storage addressed by EA+8.

If FRSp is odd, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point Double Pair Indexed X-form

\[ \text{stfdpx} \quad \text{FRSp,RA,RB} \]

| 31 | FRSp | RA | RB | 919 | /
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

If RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow |\text{RA}| \)
\( \text{EA} \leftarrow b + (\text{RB}) \)
\( \text{MEM(EA, 8)} \leftarrow \text{FRSp}_{\text{even}} \)
\( \text{MEM(EA+8, 8)} \leftarrow \text{FRSp}_{\text{odd}} \)

Let the effective address (EA) be the sum \( (\text{RA}|0) + (\text{DS||0b00}) \).

The contents of the even-numbered register of FRSp are stored into the doubleword in storage addressed by EA.
The contents of the odd-numbered register of FRSp are stored into the doubleword in storage addressed by EA+8.

If FRSp is odd, the instruction form is invalid.

Special Registers Altered:
None
4.6.5 Floating-Point Move Instructions

These instructions copy data from one floating-point register to another, altering the sign bit (bit 0) as described below for `fneg`, `fabs`, `fnabs`, and `fcpsgn`. These instructions treat NaNs just like any other kind of value (e.g., the sign bit of a NaN may be altered by `fneg`, `fabs`, `fnabs`, and `fcpsgn`). These instructions do not alter the FPSCR.

**Floating Move Register X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fmr</code> FRT,FRB</td>
<td>(Rc=0)</td>
<td>63 FRT /// FRB 72</td>
</tr>
<tr>
<td><code>fmr.</code> FRT,FRB</td>
<td>(Rc=1)</td>
<td>63 FRT /// FRB 40</td>
</tr>
</tbody>
</table>

The contents of register `FRB` are placed into register `FRT`.

Special Registers Altered:

- CR1 (if Rc=1)

**Floating Negate X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fneg</code> FRT,FRB</td>
<td>(Rc=0)</td>
<td>63 FRT /// FRB 40</td>
</tr>
<tr>
<td><code>fneg.</code> FRT,FRB</td>
<td>(Rc=1)</td>
<td>63 FRT /// FRB 136</td>
</tr>
</tbody>
</table>

The contents of register `FRB` with bit 0 inverted are placed into register `FRT`.

Special Registers Altered:

- CR1 (if Rc=1)

**Floating Absolute Value X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fabs</code> FRT,FRB</td>
<td>(Rc=0)</td>
<td>63 FRT /// FRB 264</td>
</tr>
<tr>
<td><code>fabs.</code> FRT,FRB</td>
<td>(Rc=1)</td>
<td>63 FRT /// FRB 136</td>
</tr>
</tbody>
</table>

The contents of register `FRB` with bit 0 set to zero are placed into register `FRT`.

Special Registers Altered:

- CR1 (if Rc=1)

**Floating Negative Absolute Value X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fnabs</code> FRT,FRB</td>
<td>(Rc=0)</td>
<td>63 FRT /// FRB 264</td>
</tr>
<tr>
<td><code>fnabs.</code> FRT,FRB</td>
<td>(Rc=1)</td>
<td>63 FRT /// FRB 136</td>
</tr>
</tbody>
</table>

The contents of register `FRB` with bit 0 set to one are placed into register `FRT`.

Special Registers Altered:

- CR1 (if Rc=1)

**Floating Copy Sign X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fcpsgn</code> FRT,FRA,FRB</td>
<td>(Rc=0)</td>
<td>63 FRT, FRA FRB 8</td>
</tr>
<tr>
<td><code>fcpsgn.</code> FRT,FRA,FRB</td>
<td>(Rc=1)</td>
<td>63 FRT, FRA FRB 8</td>
</tr>
</tbody>
</table>

The contents of register `FRB` with bit 0 set to the value of bit 0 of register `FRA` are placed into register `FRT`.

Special Registers Altered:

- CR1 (if Rc=1)
Floating Merge Even Word X-form

```
fmrgew FRT,FRA,FRB

if MSR.FP=0 then FP_Unavailable()
FPR[FRT].word[0] ← FPR[FRA].word[0]
FPR[FRT].word[1] ← FPR[FRB].word[0]
```

The contents of word element 0 of FPR[FRA] are placed into word element 0 of FPR[FRT].

The contents of word element 0 of FPR[FRB] are placed into word element 1 of FPR[FRT].

*fmrgew* is treated as a *Floating-Point* instruction in terms of resource availability.

**Special Registers Altered**

None

Floating Merge Odd Word X-form

```
fmrgow FRT,FRA,FRB

if MSR.FP=0 then FP_Unavailable()
FPR[FRT].word[0] ← FPR[FRA].word[1]
```

The contents of word element 1 of FPR[FRA] are placed into word element 0 of FPR[FRT].

The contents of word element 1 of FPR[FRB] are placed into word element 1 of FPR[FRT].

*fmrgow* is treated as a *Floating-Point* instruction in terms of resource availability.

**Special Registers Altered**

None
4.6.6 Floating-Point Arithmetic Instructions

4.6.6.1 Floating-Point Elementary Arithmetic Instructions

**Floating Add A-form**

```
fadd    FRT,FRA,FRB  (Rc=0)  
fadd.   FRT,FRA,FRB  (Rc=1)
```

The floating-point operand in register `FRA` is added to the floating-point operand in register `FRB`.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of `RN` and placed into register `FRT`.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (`G`, `R`, and `X`) enter into the computation.

If a carry occurs, the sum’s significand is shifted right one bit position and the exponent is increased by one.

**Floating Subtract A-form**

```
fsub    FRT,FRA,FRB  (Rc=0)  
fsub.   FRT,FRA,FRB  (Rc=1)
```

The floating-point operand in register `FRB` is subtracted from the floating-point operand in register `FRA`.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of `RN` and placed into register `FRT`.

The execution of the Floating Subtract instruction is identical to that of Floating Add, except that the contents of `FRB` participate in the operation with the sign bit (bit 0) inverted.

**Special Registers Altered:**

- `FPFR` is set to the class and sign of the result, except for Invalid Operation Exceptions when `VE=1`.
- `CR1` (if `Rc=1`)

**Floating Add Single A-form**

```
fadds   FRT,FRA,FRB  (Rc=0)  
fadds.  FRT,FRA,FRB  (Rc=1)
```

**Floating Subtract Single A-form**

```
fsubs   FRT,FRA,FRB  (Rc=0)  
fsubs.  FRT,FRA,FRB  (Rc=1)
```

**Special Registers Altered:**

- `FPFR` is set to the class and sign of the result, except for Invalid Operation Exceptions when `VE=1`.
- `CR1` (if `Rc=1`)

<table>
<thead>
<tr>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FRT</td>
<td>FRA</td>
<td>FRB</td>
<td>///</td>
<td>20</td>
<td>Rc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>59</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FRT</td>
<td>FRA</td>
<td>FRB</td>
<td>///</td>
<td>20</td>
<td>Rc</td>
</tr>
</tbody>
</table>

(If `Rc=1`)
### Floating Multiply A-form

\[
\begin{array}{llllllll}
\text{fmul} & \text{FRT,FRA,FRC} & (Rc=0) \\
\text{fmul.} & \text{FRT,FRA,FRC} & (Rc=1) \\
\end{array}
\]

The floating-point operand in register \( \text{FRA} \) is multiplied by the floating-point operand in register \( \text{FRC} \).

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of \( RN \) and placed into register \( \text{FRT} \).

Floating-point multiplication is based on exponent addition and multiplication of the significands.

\( \text{FPRF} \) is set to the class and sign of the result, except for Invalid Operation Exceptions when \( \text{VE}=1 \).

**Special Registers Altered:**

\[
\begin{array}{llll}
\text{FPRF} & \text{FR} & \text{FI} \\
\text{FX} & \text{OX} & \text{UX} & \text{XX} \\
\text{VXSNAN} & \text{VXIMZ} \\
\text{CR1} & (\text{if Rc=1}) \\
\end{array}
\]

### Floating Divide A-form

\[
\begin{array}{llllllll}
\text{fdiv} & \text{FRT,FRA,FRB} & (Rc=0) \\
\text{fdiv.} & \text{FRT,FRA,FRB} & (Rc=1) \\
\end{array}
\]

The floating-point operand in register \( \text{FRA} \) is divided by the floating-point operand in register \( \text{FRB} \). The remainder is not supplied as a result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of \( RN \) and placed into register \( \text{FRT} \).

Floating-point division is based on exponent subtraction and division of the significands.

\( \text{FPRF} \) is set to the class and sign of the result, except for Invalid Operation Exceptions when \( \text{VE}=1 \) and Zero Divide Exceptions when \( \text{ZE}=1 \).

**Special Registers Altered:**

\[
\begin{array}{llll}
\text{FPRF} & \text{FR} & \text{FI} \\
\text{FX} & \text{OX} & \text{UX} & \text{ZX} & \text{XX} \\
\text{VXSNAN} & \text{VXIDI} & \text{VXZDZ} \\
\text{CR1} & (\text{if Rc=1}) \\
\end{array}
\]
Floating Square Root A-form

\[
\text{fsqrt} \quad \text{FRT,FRB} \quad (Rc=0) \\
\text{fsqrt} \cdot \quad \text{FRT,FRB} \quad (Rc=1)
\]

Floating Square Root Single A-form

\[
\text{fsqrts} \quad \text{FRT,FRB} \quad (Rc=0) \\
\text{fsqrts} \cdot \quad \text{FRT,FRB} \quad (Rc=1)
\]

The square root of the floating-point operand in register \( FRB \) is placed into register \( FRT \).

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of \( RN \) and placed into register \( FRT \).

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>(&lt; 0)</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>(0)</td>
<td>(-0)</td>
<td>None</td>
</tr>
<tr>
<td>(\infty)</td>
<td>(\infty)</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN(^1)</td>
<td>VXSQNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

\(^1\) No result if \( VE=1 \)

FPSCR\(_{FFP} \) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\(_{VE} = 1 \).

Special Registers Altered:

FPFR F R FI FX OX UX XX
VXSQRT VXSQRT
CR1 (if \( Rc=1 \))

Floating Reciprocal Estimate A-form

\[
\text{fre} \quad \text{FRT,FRB} \quad (Rc=0) \\
\text{fre} \cdot \quad \text{FRT,FRB} \quad (Rc=1)
\]

Floating Reciprocal Estimate Single A-form

\[
\text{fres} \quad \text{FRT,FRB} \quad (Rc=0) \\
\text{fres} \cdot \quad \text{FRT,FRB} \quad (Rc=1)
\]

An estimate of the reciprocal of the floating-point operand in register \( FRB \) is placed into register \( FRT \). Unless the reciprocal would be a zero, an infinity, the result of a trap-disabled Overflow exception, or a QNaN, the estimate is correct to a precision of one part in 256 of the reciprocal of \( (FRB) \), i.e.,

\[
\text{ABS} \left( \frac{\text{estimate} - 1/x}{1/x} \right) \leq \frac{1}{256}
\]

where \( x \) is the initial value in \( FRB \).

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\frac{1}{2})</td>
<td>(-0)</td>
<td>None</td>
</tr>
<tr>
<td>(0)</td>
<td>(-\frac{1}{2})</td>
<td>ZK</td>
</tr>
<tr>
<td>(+0)</td>
<td>(+\frac{1}{2})</td>
<td>ZK</td>
</tr>
<tr>
<td>(+\frac{1}{2})</td>
<td>(+0)</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN(^2)</td>
<td>VXSQNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

\(^1\) No result if \( ZE=1 \).
\(^2\) No result if \( VE=1 \).

FPFR is set to the class and sign of the result, except for Invalid Operation Exceptions when \( VE=1 \) and Zero Divide Exceptions when \( ZE=1 \).

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

FPFR F R (undefined) F I (undefined)
FX OX UX ZK XX (undefined)
VXSQNAN
CR1 (if \( Rc=1 \))

Programming Note

For the Floating-Point Estimate instructions, some implementations might implement a precision higher than the minimum architected precision. Thus, a program may take advantage of the higher precision instructions to increase performance by decreasing the iterations needed for software emulation of floating-point instructions. However, there is no guarantee given about the precision which may vary (up or down) between implementations. Only programs targeted at a specific implementation (i.e., the program will not be migrated to another implementation) should take advantage of the higher precision of the instructions. All other programs should rely on the minimum architected precision, which will guarantee the program to run properly across different implementations.
**Floating Reciprocal Square Root Estimate A-form**

frsqrte FRT,FRB (Rc=0)
frsqrte FRT,FRB (Rc=1)

A estimate of the reciprocal of the square root of the floating-point operand in register \( FRB \) is placed into register \( FRT \). The estimate placed into register \( FRT \) is correct to a precision of one part in 32 of the reciprocal of \( \sqrt{FRB} \), i.e.,

\[
\text{ABS} \left( \frac{1}{\sqrt{x}} - \frac{1}{\sqrt{x}} \right) \leq \frac{1}{32}
\]

where \( x \) is the initial value in \( FRB \).

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>QNaN (^2)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>(&lt; 0)</td>
<td>QNaN (^2)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>(-0)</td>
<td>(-\infty)</td>
<td>ZK</td>
</tr>
<tr>
<td>(+0)</td>
<td>(+\infty)</td>
<td>ZK</td>
</tr>
<tr>
<td>(\pm\infty)</td>
<td>(\pm0)</td>
<td>None</td>
</tr>
<tr>
<td>SN NaN</td>
<td>QNaN (^2)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

1 No result if \( VE=1 \).
2 No result if \( VE=1 \).

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when \( VE=1 \) and Zero Divide Exceptions when \( VE=1 \).

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

**Special Registers Altered:**
- FPRF: FR (undefined) F1 (undefined)
- FX: OX UX ZK XX (undefined)
- VXSQRT: VXSQRT
- CR1: CR1 (if \( Rc=1 \))

**Note**

See the Notes that appear with \( fre[s] \).

---

**Floating Test for software Divide X-form**

\( \text{ftdiv BF,FRA,FRB} \)

Let \( e_a \) be the unbiased exponent of the double-precision floating-point operand in register \( FRA \).

Let \( e_b \) be the unbiased exponent of the double-precision floating-point operand in register \( FRB \).

\( fe \_\text{flag} \) is set to 1 if any of the following conditions occurs.

- The double-precision floating-point operand in register \( FRA \) is a NaN or an Infinity.
- The double-precision floating-point operand in register \( FRB \) is a Zero, a NaN, or an Infinity.
- \( e_b \) is less than or equal to \(-1022\).
- \( e_b \) is greater than or equal to \(1021\).
- The double-precision floating-point operand in register \( FRA \) is not a zero and the difference, \( e_a - e_b \), is greater than or equal to \(1023\).
- The double-precision floating-point operand in register \( FRA \) is not a zero and the difference, \( e_a - e_b \), is less than or equal to \(-1021\).
- The double-precision floating-point operand in register \( FRA \) is not a zero and \( e_a \) is less than or equal to \(-970\).

Otherwise \( fe \_\text{flag} \) is set to 0.

\( fg \_\text{flag} \) is set to 1 if either of the following conditions occurs.

- The double-precision floating-point operand in register \( FRA \) is an Infinity.
- The double-precision floating-point operand in register \( FRB \) is a Zero, an Infinity, or a denormalized value.

Otherwise \( fg \_\text{flag} \) is set to 0.

If the implementation guarantees a relative error of \( fre[s] \) of less than or equal to \(2^{-14} \), then \( fl \_\text{flag} \) is set to 1. Otherwise \( fl \_\text{flag} \) is set to 0.

**CR field BF** is set to the value \( \text{fl}_\_\text{flag} || \text{fg}_\_\text{flag} || \text{fe}_\_\text{flag} || 0b0 \).

**Special Registers Altered:**
- CR field BF
Floating Test for software Square Root
X-form

ftsqrt BF,FRB

<table>
<thead>
<tr>
<th>63</th>
<th>BF</th>
<th>//</th>
<th>///</th>
<th>FRB</th>
<th>160</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

Let \( e_b \) be the unbiased exponent of the double-precision floating-point operand in register \( FRB \).

\( fe\_flag \) is set to 1 if either of the following conditions occurs.

- The double-precision floating-point operand in register \( FRB \) is a zero, a NaN, or an infinity, or a negative value.
- \( e_b \) is less than or equal to \(-970\).

Otherwise \( fe\_flag \) is set to 0.

\( fg\_flag \) is set to 1 if the following condition occurs.

- The double-precision floating-point operand in register \( FRB \) is a Zero, an Infinity, or a denormalized value.

Otherwise \( fg\_flag \) is set to 0.

If the implementation guarantees a relative error of \( frsqrte[\text{s}][\text{u}] \) of less than or equal to \( 2^{-14} \), then \( fl\_flag \) is set to 1. Otherwise \( fl\_flag \) is set to 0.

CR field BF is set to the value \( fl\_flag \| fg\_flag \| fe\_flag \| 0b0 \).

Special Registers Altered:
CR field BF

Programming Note

\texttt{ftdiv} and \texttt{ftsqrt} are provided to accelerate software emulation of divide and square root operations, by performing the requisite special case checking. Software needs only a single branch, on FE=1 (in CR[BF]), to a special case handler. FG and FL may provide further acceleration opportunities.
4.6.6.2 Floating-Point Multiply-Add Instructions

These instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide (L bit, FRACTION), and all 106 bits take part in the add/subtract portion of the instruction.

Status bits are set as follows.

- Overflow, Underflow, and Inexact Exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, and not on the result of the multiplication.
- Invalid Operation Exception bits are set as if the multiplication and the addition were performed using two separate instructions (fmul[s], followed by fadd[s] or fsub[s]). That is, multiplication of infinity by 0 or of anything by an SNaN, and/or addition of an SNaN, cause the corresponding exception bits to be set.

Floating Multiply-Add A-form

| fmadd | FRT,FRA,FRC,FRB  | (Rc=0) |
| fmadd. | FRT,FRA,FRC,FRB  | (Rc=1) |
| 63 | FRT | FRA | FRB | FRC | 29 | % |
| 0 | 6 | 11 | 16 | 21 | 35 | 31 |

The operation

\[ FRT ← [(FRA)´(FRC)] + (FRB) \]

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN and placed into register FRT.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

| FPRF | FR | FI |
| FX | OX | UX | XX |
| VXSNaN | VXISI | VXIMZ |
CR1 (if Rc=1)

Floating Multiply-Add Single A-form

| fmadds | FRT,FRA,FRC,FRB  | (Rc=0) |
| fmadds. | FRT,FRA,FRC,FRB  | (Rc=1) |
| 59 | FRT | FRA | FRB | FRC | 29 | % |
| 0 | 6 | 11 | 16 | 21 | 35 | 31 |

The operation

\[ FRT ← [(FRA)´(FRC)] + (FRB) \]

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN and placed into register FRT.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

| FPRF | FR | FI |
| FX | OX | UX | XX |
| VXSNaN | VXISI | VXIMZ |
CR1 (if Rc=1)

Floating Multiply-Subtract A-form

| fmsub | FRT,FRA,FRC,FRB  | (Rc=0) |
| fmsub. | FRT,FRA,FRC,FRB  | (Rc=1) |
| 63 | FRT | FRA | FRB | FRC | 28 | % |
| 0 | 6 | 11 | 16 | 21 | 35 | 31 |

The operation

\[ FRT ← [(FRA)´(FRC)] - (FRB) \]

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN and placed into register FRT.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

| FPRF | FR | FI |
| FX | OX | UX | XX |
| VXSNaN | VXISI | VXIMZ |
CR1 (if Rc=1)

Floating Multiply-Subtract Single A-form

| fmsubs | FRT,FRA,FRC,FRB  | (Rc=0) |
| fmsubs. | FRT,FRA,FRC,FRB  | (Rc=1) |
| 59 | FRT | FRA | FRB | FRC | 28 | % |
| 0 | 6 | 11 | 16 | 21 | 35 | 31 |

The operation

\[ FRT ← [(FRA)´(FRC)] - (FRB) \]

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN and placed into register FRT.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

| FPRF | FR | FI |
| FX | OX | UX | XX |
| VXSNaN | VXISI | VXIMZ |
CR1 (if Rc=1)
Floating Negative Multiply-Add A-form

\[
\text{fnmadd} \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=0) \\
\text{fnmadd.} \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=1)
\]

The operation
\[
\text{FRT} \leftarrow - ((\text{FRA} \cdot \text{FRC}) + \text{FRB})
\]
is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the Floating Multiply-Add instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

\[
\begin{array}{ccccccc}
\text{FPRF} & \text{FR} & \text{FI} \\
\text{FX} & \text{OX} & \text{UX} & \text{XX} \\
\text{VXSNAN} & \text{VXI} & \text{SI} & \text{VXI} & \text{MZ} \\
\text{CR1} & & & & & \\
\end{array}
\]

\[(\text{if Rc=1})\]

Floating Negative Multiply-Add Single A-form

\[
\text{fnmadds} \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=0) \\
\text{fnmadds.} \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=1)
\]

Floating Negative Multiply-Subtract A-form

\[
\text{fnmsub} \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=0) \\
\text{fnmsub.} \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=1)
\]

Floating Negative Multiply-Subtract Single A-form

\[
\text{fnmsubs} \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=0) \\
\text{fnmsubs.} \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=1)
\]

The operation
\[
\text{FRT} \leftarrow - ((\text{FRA} \cdot \text{FRC}) - \text{FRB})
\]
is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the Floating Multiply-Subtract instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

\[
\begin{array}{ccccccc}
\text{FPRF} & \text{FR} & \text{FI} \\
\text{FX} & \text{OX} & \text{UX} & \text{XX} \\
\text{VXSNAN} & \text{VXI} & \text{SI} & \text{VXI} & \text{MZ} \\
\text{CR1} & & & & & \\
\end{array}
\]

\[(\text{if Rc=1})\]
4.6.7 Floating-Point Rounding and Conversion Instructions

4.6.7.1 Floating-Point Rounding Instruction

Floating Round to Single-Precision X-form

\[
\begin{array}{c|cccccc}
\text{frsp} & \text{FRB} & \text{Rc} = 0 & \text{FRT} & \text{Rc} = 1 & \text{FRB} & \text{FRT} \\
\hline
0 & 63 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

The floating-point operand in register \( \text{FRB} \) is rounded to single-precision, using the rounding mode specified by \( \text{RN} \), and placed into register \( \text{FRT} \).

The rounding is described fully in Section A.1, “Floating-Point Round to Single-Precision Model” on page 995.

\( \text{FPRF} \) is set to the class and sign of the result, except for Invalid Operation Exceptions when \( \text{VE}=1 \).

Special Registers Altered:

- \( \text{FPRF} \) (undefined)
- \( \text{FR} \)
- \( \text{FI} \)
- \( \text{FX} \)
- \( \text{XX} \)
- \( \text{VXSNAN} \)
- \( \text{VXCVI} \)
- \( \text{CR1} \) (if \( \text{Rc}=1 \))

4.6.7.2 Floating-Point Convert To/From Integer Instructions

Floating Convert with round Double-Precision To Signed Doubleword format X-form

\[
\begin{array}{c|cccccc}
\text{fctid} & \text{FRB} & \text{Rc} = 0 & \text{FRT} & \text{Rc} = 1 & \text{FRB} & \text{FRT} \\
\hline
0 & 63 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

Let \( \text{src} \) be the double-precision floating-point value in \( \text{FRB} \).

If \( \text{src} \) is a NaN, then the result is \( 0x8000_0000_0000_0000 \), \( \text{VXCVI} \) is set to 1, and, if \( \text{src} \) is an SNaN, \( \text{VXSNAN} \) is set to 1.

Otherwise, \( \text{src} \) is rounded to a floating-point integer using the rounding mode specified by \( \text{RN} \).

If the rounded value is greater than \( 2^{63} \), then the result is \( 0x7FFF_FFFF_FFFF_FFFF \) and \( \text{VXCVI} \) is set to 1.

Otherwise, if the rounded value is less than \( -2^{63} \), then the result is \( 0x8000_0000_0000_0000 \) and \( \text{VXCVI} \) is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and \( \text{XX} \) is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into \( \text{FRT} \).

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 999.

Except for enabled Invalid Operation Exceptions, \( \text{FPRF} \) is undefined. \( \text{FR} \) is set if the result is incremented when rounded. \( \text{FI} \) is set if the result is inexact.

Special Registers Altered:

- \( \text{FPRF} \) (undefined)
- \( \text{FR} \)
- \( \text{FI} \)
- \( \text{FX} \)
- \( \text{XX} \)
- \( \text{VXSNAN} \)
- \( \text{VXCVI} \)
- \( \text{CR1} \) (if \( \text{Rc}=1 \))
Range Convert with truncate
Double-Precision To Signed Doubleword format X-form

\[ fctidz \text{ FRT,FRB (Rc=0)} \]
\[ fctidz. \text{ FRT,FRB (Rc=1)} \]

<table>
<thead>
<tr>
<th>63</th>
<th>FRT</th>
<th>///</th>
<th>FRB</th>
<th>815</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

Let \( src \) be the double-precision floating-point value in \( FRB \).

If \( src \) is a NaN, then the result is \( 0x8000_0000_0000_0000 \), \( VXCVI \) is set to 1, and, if \( src \) is an SNaN, \( VXSNAN \) is set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than \( 2^{63} - 1 \), then the result is \( 0x7FFF_FFFF_FFFF_FFFF \), and \( VXCVI \) is set to 1.

Otherwise, if the rounded value is less than \( -2^{63} \), then the result is \( 0x8000_0000_0000_0000 \), and \( VXCVI \) is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and \( XX \) is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into \( FRT \).

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 999.

Except for enabled Invalid Operation Exceptions, \( FPRF \) is undefined. \( FR \) is set if the result is incremented when rounded. \( FI \) is set if the result is inexact.

Special Registers Altered:
- \( FPRF \) (undefined)
- \( FR \)
- \( FI \)
- \( FX XX VXSNAN VXCVI \)
- \( CR1 \) (if \( Rc=1 \))

Range Convert with round Double-Precision To Unsigned Doubleword format X-form

\[ fctidu \text{ FRT,FRB (Rc=0)} \]
\[ fctidu. \text{ FRT,FRB (Rc=1)} \]

<table>
<thead>
<tr>
<th>63</th>
<th>FRT</th>
<th>///</th>
<th>FRB</th>
<th>942</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>96</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

Let \( src \) be the double-precision floating-point value in \( FRB \).

If \( src \) is a NaN, then the result is \( 0x0000_0000_0000_0000 \), \( VXCVI \) is set to 1, and, if \( src \) is an SNaN, \( VXSNAN \) is set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode specified by \( RN \).

If the rounded value is greater than \( 2^{64} - 1 \), then the result is \( 0xFFFF_FFFF_FFFF_FFFF \), and \( VXCVI \) is set to 1.

Otherwise, if the rounded value is less than 0, then the result is \( 0x0000_0000_0000_0000 \), and \( VXCVI \) is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format, and \( XX \) is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into \( FRT \).

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 999.

Except for enabled Invalid Operation Exceptions, \( FPRF \) is undefined. \( FR \) is set if the result is incremented when rounded. \( FI \) is set if the result is inexact.

Special Registers Altered:
- \( FPRF \) (undefined)
- \( FR \)
- \( FI \)
- \( FX XX VXSNAN VXCVI \)
- \( CR1 \) (if \( Rc=1 \))
Floating Convert with truncate
Double-Precision To Unsigned Doubleword format X-form

\[
\begin{array}{c|c|c|c|c|c|c}
 & FRT & FRB & & & & \\
\hline 63 & 6 & 11 & 16 & 21 & 943 & \%
\hline 0 & & & & & & \end{array}
\]

Let \( \text{src} \) be the double-precision floating-point value in \( \text{FRB} \).

If \( \text{src} \) is a NaN, then the result is \( 0x0000_0000_0000_0000 \), \( \text{VXCVI} \) is set to 1, and, if \( \text{src} \) is an SNaN, \( \text{VXSNAN} \) is set to 1.

Otherwise, \( \text{src} \) is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than \( 2^{64} - 1 \), then the result is \( 0xFFFF_FFFF_FFFF_FFFF \), and \( \text{VXCVI} \) is set to 1.

Otherwise, if the rounded value is less than 0, then the result is \( 0x0000_0000_0000_0000 \), and \( \text{VXCVI} \) is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format, and \( \text{XX} \) is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into \( \text{FRT} \).

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 999.

Except for enabled Invalid Operation Exceptions, \( \text{FPRF} \) is undefined. \( \text{FR} \) is set if the result is incremented when rounded. \( \text{FI} \) is set if the result is inexact.

Special Registers Altered:
\[
\begin{align*}
\text{FPRF} & \quad \text{(undefined)} \\
\text{FR} & \quad \text{FI} \\
\text{FX} & \quad \text{VXSNAN} \\
\text{VXCVI} & \\
\text{CR1} & \quad \text{(if } \text{RC}=1) 
\end{align*}
\]

Floating Convert with round Double-Precision To Signed Word format X-form

\[
\begin{array}{c|c|c|c|c|c|c}
 & FRT & FRB & & & & \\
\hline 63 & 6 & 11 & 16 & 21 & 14 & \%
\hline 0 & & & & & & \end{array}
\]

Let \( \text{src} \) be the double-precision floating-point value in \( \text{FRB} \).

If \( \text{src} \) is a NaN, then the result is \( 0x8000_0000 \), \( \text{VXCVI} \) is set to 1, and, if \( \text{src} \) is an SNaN, \( \text{VXSNAN} \) is set to 1.

Otherwise, \( \text{src} \) is rounded to a floating-point integer using the rounding mode specified by \( \text{RN} \).

If the rounded value is greater than \( 2^{31} - 1 \), then the result is \( 0x7FFF_FFFF \), and \( \text{VXCVI} \) is set to 1.

Otherwise, if the rounded value is less than \( -2^{31} \), then the result is \( 0x8000_0000 \), and \( \text{VXCVI} \) is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format, and \( \text{XX} \) is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into \( \text{FRT} \) and \( \text{FRT} \) is undefined.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 999.

Except for enabled Invalid Operation Exceptions, \( \text{FPRF} \) is undefined. \( \text{FR} \) is set if the result is incremented when rounded. \( \text{FI} \) is set if the result is inexact.

Special Registers Altered:
\[
\begin{align*}
\text{FPRF} & \quad \text{(undefined)} \\
\text{FR} & \quad \text{FI} \\
\text{FX} & \quad \text{VXSNAN} \\
\text{VXCVI} & \\
\text{CR1} & \quad \text{(if } \text{RC}=1) 
\end{align*}
\]
Floating Convert with truncate
Double-Precision To Signed Word format X-form

Let \( \text{src} \) be the double-precision floating-point value in \( \text{FRB} \).

If \( \text{src} \) is a NaN, then the result is \( 0x8000_0000 \), \( \text{VXCVI} \) is set to 1, and, if \( \text{src} \) is an SNaN, \( \text{VXSNAN} \) is set to 1.

Otherwise, \( \text{src} \) is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than \( 2^{31} - 1 \), then the result is \( 0x7FFF_FFFF \), and \( \text{VXCVI} \) is set to 1.

Otherwise, if the rounded value is less than \( -2^{31} \), then the result is \( 0x8000_0000 \), and \( \text{VXCVI} \) is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format, and \( \text{XX} \) is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into \( \text{FRT}_{32:63} \) and \( \text{FRT}_{0:31} \) is undefined.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 999.

Except for enabled Invalid Operation Exceptions, \( \text{FPRF} \) is undefined. \( \text{FR} \) is set if the result is incremented when rounded. \( \text{FI} \) is set if the result is inexact.

Special Registers Altered:

\[ \begin{align*}
&\text{FPRF} \quad \text{(undefined)} \\
&\text{FR} \\
&\text{FI} \\
&\text{FX} \\
&\text{XX} \\
&\text{VXSNAN} \\
&\text{VXCVI} \\
&\text{CR1} \\
\end{align*} \]

\( \text{(if } \text{Rc}=1 \text{) } \)

Floating Convert with round Double-Precision
To Unsigned Word format X-form

Let \( \text{src} \) be the double-precision floating-point value in \( \text{FRB} \).

If \( \text{src} \) is a NaN, then the result is \( 0x0000_0000 \), \( \text{VXCVI} \) is set to 1, and, if \( \text{src} \) is an SNaN, \( \text{VXSNAN} \) is set to 1.

Otherwise, \( \text{src} \) is rounded to a floating-point integer using the rounding mode specified by \( \text{RN} \).

If the rounded value is greater than \( 2^{32} - 1 \), then the result is \( 0xFFF_FFFF \) and \( \text{VXCVI} \) is set to 1.

Otherwise, if the rounded value is less than 0, then the result is \( 0x0000_0000 \) and \( \text{VXCVI} \) is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and \( \text{XX} \) is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into \( \text{FRT}_{32:63} \) and \( \text{FRT}_{0:31} \) is undefined.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 999.

Except for enabled Invalid Operation Exceptions, \( \text{FPRF} \) is undefined. \( \text{FR} \) is set if the result is incremented when rounded. \( \text{FI} \) is set if the result is inexact.

Special Registers Altered:

\[ \begin{align*}
&\text{FPRF} \quad \text{(undefined)} \\
&\text{FR} \\
&\text{FI} \\
&\text{FX} \\
&\text{XX} \\
&\text{VXSNAN} \\
&\text{VXCVI} \\
&\text{CR1} \\
\end{align*} \]

\( \text{(if } \text{Rc}=1 \text{) } \)
Floating Convert with truncate
Double-Precision To Unsigned Word format
X-form

\[ \text{fctiwuz, FRT,FRB} \quad (Rc=0) \]
\[ \text{fctiwuz, FRT,FRB} \quad (Rc=1) \]

Let \( src \) be the double-precision floating-point value in \( FRB \).

If \( src \) is a NaN, then the result is 0x0000_0000, \( VXCVI \) is set to 1, and, if \( src \) is an SNaN, \( VXSNAN \) is set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than \( 2^{32} - 1 \), then the result is 0xFFFF_FFFF and \( VXCVI \) is set to 1.

Otherwise, if the rounded value is less than 0.0, then the result is 0x0000_0000 and \( VXCVI \) is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and \( XX \) is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into \( FRT_{32:63} \) and \( FRT_{0:31} \) is undefined.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 999.

Except for enabled Invalid Operation Exceptions, \( FPRF \) is undefined. \( FR \) is set if the result is incremented when rounded. \( FI \) is set if the result is inexact.

Special Registers Altered:

- \( FPRF \) (undefined)
- \( FR \)
- \( FI \)
- \( FX \)
- \( XX \)
- \( VXSNAN \)
- \( VXCVI \)
- \( CR1 \) (if \( Rc=1 \))

Floating Convert with round Signed
Doubleword to Double-Precision format
X-form

\[ \text{fcfid, FRT,FRB} \quad (Rc=0) \]
\[ \text{fcfid, FRT,FRB} \quad (Rc=1) \]

The 64-bit signed fixed-point operand in register \( FRB \) is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, using the rounding mode specified by \( RN \), and placed into register \( FRT \).

The conversion is described fully in Section A.3, “Floating-Point Convert from Integer Model”.

\( FPRF \) is set to the class and sign of the result. \( FR \) is set if the result is incremented when rounded. \( FI \) is set if the result is inexact.

Special Registers Altered:

- \( FPRF \)
- \( FR \)
- \( FI \)
- \( FX \)
- \( XX \)
- \( CR1 \) (if \( Rc=1 \))

--- Programming Note ---

Converting a signed integer word to double-precision floating-point can be accomplished by loading the word from storage using Load Float Word Algebraic Indexed and then using \( fcfid \).
**Floating Convert with round Unsigned Doubleword to Double-Precision format X-form**

fcfidu  FRT,FRB  \( (Rc=0) \)
fcfidu. FRT,FRB  \( (Rc=1) \)

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>FRT</th>
<th>11</th>
<th>FRB</th>
<th>974</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

The 64-bit unsigned fixed-point operand in register \( FRB \) is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, using the rounding mode specified by \( RN \), and placed into register \( FRT \).

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

**FPRF** is set to the class and sign of the result. \( FR \) is set if the result is incremented when rounded. \( FI \) is set if the result is inexact.

**Special Registers Altered:**

- \( FPRF \)
- \( FR \)
- \( FI \)
- \( FX \)
- \( XX \)
- \( CR1 \)

**Programming Note**

Converting an unsigned integer word to double-precision floating-point can be accomplished by loading the word from storage using \textit{Load Float Word and Zero Indexed} and then using \textit{fcfidu}.

---

**Floating Convert with round Signed Doubleword to Single-Precision format X-form**

fcfids  FRT,FRB  \( (Rc=0) \)
fcfids. FRT,FRB  \( (Rc=1) \)

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>FRT</th>
<th>11</th>
<th>FRB</th>
<th>846</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

The 64-bit signed fixed-point operand in register \( FRB \) is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to single-precision, using the rounding mode specified by \( RN \), and placed into register \( FRT \).

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

**FPRF** is set to the class and sign of the result. \( FR \) is set if the result is incremented when rounded. \( FI \) is set if the result is inexact.

**Special Registers Altered:**

- \( FPRF \)
- \( FR \)
- \( FI \)
- \( FX \)
- \( XX \)
- \( CR1 \)

**Programming Note**

Converting a signed integer word to single-precision floating-point can be accomplished by loading the word from storage using \textit{Load Float Word Algebraic Indexed} and then using \textit{fcfids}.
**Floating Convert with round Unsigned Doubleword to Single-Precision format X-form**

```
fcfidus. FRT,FRB (Rc = 1)
```

The 64-bit unsigned fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to single-precision, using the rounding mode specified by RN, and placed into register FRT.

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

FPRF is set to the class and sign of the result. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

**Special Registers Altered:**

- FPRF
- FR
- FI
- FX
- XX
- CR1 (if Rc = 1)

**Programming Note**

Converting an unsigned integer word to single-precision floating-point can be accomplished by loading the word from storage using `Load Float Word and Zero Indexed` and then using `fcfidus`.
4.6.7.3 Floating Round to Integer Instructions

The Floating Round to Integer instructions provide direct support for rounding functions found in high level languages. For example, `frin`, `friz`, `frip`, and `frim` implement C++ `round()`, `trunc()`, `ceil()`, and `floor()`, respectively. Note that `frin` does not implement the IEEE Round to Nearest function, which is often further described as “ties to even.” The rounding performed by these instructions is described fully in Section A.4, “Floating-Point Round to Integer Model” on page 1004.

Programming Note

These instructions set FR and FI to 0b00 regardless of whether the result is inexact or rounded because there is a desire to preserve the value of XX. Furthermore, it is believed that most programs do not need to know whether these rounding operations produce inexact or rounded results. If it is necessary to determine whether the result is inexact or rounded, software must compare the result with the original source operand.
### Floating Round to Integer Nearest X-form

<table>
<thead>
<tr>
<th></th>
<th>FRT,FRB</th>
<th>(Rc = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td>11</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRB is rounded to an integral value as follows, with the result placed into register FRT. If the sign of the operand is positive, \((FRB) + 0.5\) is truncated to an integral value, otherwise \((FRB) - 0.5\) is truncated to an integral value.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE = 1.

**Special Registers Altered:**
- FPRF: FR (set to 0) FI (set to 0)
- FX
- VXSNAN
- CR1 (if Rc = 1)

### Floating Round to Integer Toward Zero X-form

<table>
<thead>
<tr>
<th></th>
<th>FRT,FRB</th>
<th>(Rc = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td>11</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward zero, and the result is placed into register FRT. FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE = 1.

**Special Registers Altered:**
- FPRF: FR (set to 0) FI (set to 0)
- FX
- VXSNAN
- CR1 (if Rc = 1)

### Floating Round to Integer Plus X-form

<table>
<thead>
<tr>
<th></th>
<th>FRT,FRB</th>
<th>(Rc = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td>11</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward +infinity, and the result is placed into register FRT. FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE = 1.

**Special Registers Altered:**
- FPRF: FR (set to 0) FI (set to 0)
- FX
- VXSNAN
- CR1 (if Rc = 1)

### Floating Round to Integer Minus X-form

<table>
<thead>
<tr>
<th></th>
<th>FRT,FRB</th>
<th>(Rc = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td>11</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward -infinity, and the result is placed into register FRT. FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE = 1.

**Special Registers Altered:**
- FPRF: FR (set to 0) FI (set to 0)
- FX
- VXSNAN
- CR1 (if Rc = 1)
4.6.8 Floating-Point Compare Instructions

The floating-point Compare instructions compare the contents of two floating-point registers. Comparison ignores the sign of zero (i.e., regards +0 as equal to −0). The comparison can be ordered or unordered.

The comparison sets one bit in the designated CR field to 1 and the other three to 0. The FPCC is set in the same way.

**Floating Compare Unordered X-form**

\[
\text{fcmpu BF,FRA,FRB}
\]

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(FRA) &lt; (FRB)</td>
</tr>
<tr>
<td>1</td>
<td>(FRA) &gt; (FRB)</td>
</tr>
<tr>
<td>2</td>
<td>(FRA) = (FRB)</td>
</tr>
<tr>
<td>3</td>
<td>(FRA) ? (FRB) (unordered)</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, then VXSNAN is set.

**Special Registers Altered:**
- CR field BF
- FPCC
- FX
- VXSNAN

**Floating Compare Ordered X-form**

\[
\text{fcmpo BF,FRA,FRB}
\]

The CR field and the FPCC are set as follows.

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(FRA) &lt; (FRB)</td>
</tr>
<tr>
<td>1</td>
<td>(FRA) &gt; (FRB)</td>
</tr>
<tr>
<td>2</td>
<td>(FRA) = (FRB)</td>
</tr>
<tr>
<td>3</td>
<td>(FRA) ? (FRB) (unordered)</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, then VXSNAN is set.

If Invalid Operation is disabled (VE=0), VXVC is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, then VXVC is set.

**Special Registers Altered:**
- CR field BF
- FPCC
- FX
- VXSNAN VXVC
4.6.9 Floating-Point Select Instruction

**Floating Select A-form**

.fsella FRT,FRA,FRC,FRB (Rc=0)
.fsela FRT,FRA,FRC,FRB (Rc=1)

<table>
<thead>
<tr>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FRT</td>
<td>FRA</td>
<td>FRB</td>
<td>FRC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if (FRA) \( \geq 0.0 \) then FRT \( \leftarrow \) (FRC)
else FRT \( \leftarrow \) (FRB)

The floating-point operand in register FRA is compared to the value zero. If the operand is greater than or equal to zero, register FRT is set to the contents of register FRC. If the operand is less than zero or is a NaN, register FRT is set to the contents of register FRB. The comparison ignores the sign of zero (i.e., regards +0 as equal to −0).

**Special Registers Altered:**

CR1 (if Rc = 1)

---

**Programming Note**

**Warning:** Care must be taken in using fsel if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.
**fsel Usage Notes**

This section gives examples of how the *Floating Select* instruction can be used to implement certain simple forms of if-then-else constructions, without branching.

The examples show program fragments in an imaginary, C-like, high-level programming language, and the corresponding program fragment using *fsel* and other Power ISA instructions. In the examples, \(a, b, x, y,\) and \(z\) are floating-point variables, which are assumed to be in FPRs \(fa, fb, fx, fy,\) and \(fz\). FPR \(fs\) is assumed to be available for scratch space.

**Warning:** Care must be taken in using *fsel* if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section ...

### Comparison to Zero

<table>
<thead>
<tr>
<th>High-level language:</th>
<th>Power ISA:</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a \geq 0.0) then (x \leftarrow y) else (x \leftarrow z)</td>
<td><em>fsel</em> (fx, fa, fy, fz)</td>
<td>(1)</td>
</tr>
<tr>
<td>if (a &gt; 0.0) then (x \leftarrow y) else (x \leftarrow z)</td>
<td>(fneg) (fs, fa) <em>fsel</em> (fx, fs, fz, fy)</td>
<td>(1, 2)</td>
</tr>
<tr>
<td>if (a = 0.0) then (x \leftarrow y) else (x \leftarrow z)</td>
<td><em>fsel</em> (fx, fa, fy, fz) (fneg) (fs, fa) <em>fsel</em> (fx, fs, fz, fz)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

### Simple if-then-else Constructions

<table>
<thead>
<tr>
<th>High-level language:</th>
<th>Power ISA:</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a \geq b) then (x \leftarrow y) else (x \leftarrow z)</td>
<td><em>fsub</em> (fs, fa, fb) <em>fsel</em> (fx, fs, fy, fz)</td>
<td>(4, 5)</td>
</tr>
<tr>
<td>if (a &gt; b) then (x \leftarrow y) else (x \leftarrow z)</td>
<td>(fneg) (fs, fs) (fneg) (fs, fs) <em>fsel</em> (fx, fs, fx, fz)</td>
<td>(3, 4, 5)</td>
</tr>
<tr>
<td>if (a = b) then (x \leftarrow y) else (x \leftarrow z)</td>
<td>(fneg) (fs, fs) (fneg) (fs, fs) <em>fsel</em> (fx, fs, fx, fz)</td>
<td>(4, 5)</td>
</tr>
</tbody>
</table>

**Notes:**

The following Notes apply to the preceding examples and to the corresponding cases using the other three arithmetic relations \(<, \leq,\) and \(!=\). They should also be considered when any other use of *fsel* is contemplated.

In these Notes, the “optimized program” is the Power ISA program shown, and the "unoptimized program" (not shown) is the corresponding Power ISA program that uses *fcmpeq* and *Branch Conditional* instructions instead of *fsel*.

1. The unoptimized program affects the \(VXSNAN\) bit of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exception is enabled, while the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE standard.

2. The optimized program gives the incorrect result if \(a\) is a NaN.

3. The optimized program gives the incorrect result if \(a\) and/or \(b\) is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).

4. The optimized program gives the incorrect result if \(a\) and \(b\) are infinities of the same sign. (Here it is assumed that Invalid Operation Exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if Invalid Operation Exceptions are enabled, because in that case the target register of the subtraction is unchanged.)

5. The optimized program affects the \(OX, UX, XX,\) and \(VXSI\) bits of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled, while the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE standard.
4.6.10 Floating-Point Status and Control Register Instructions

Except as described below for \( \text{mffsce, mffscdrn}[i] \), \( \text{mffscrn}[i] \), and \( \text{mffsl} \), Floating-Point Status and Control Register instructions synchronize the effects of all floating-point instructions executed by a given processor. Executing a Floating-Point Status and Control Register instruction ensures that all floating-point instructions previously initiated by the given processor have completed before the Floating-Point Status and Control Register instruction is initiated, and that no subsequent floating-point instructions are initiated by the given processor until the Floating-Point Status and Control Register instruction has completed. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the Floating-Point Status and Control Register instruction is initiated.
- All invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated instructions have occurred before the Floating-Point Status and Control Register instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits is initiated until the Floating-Point Status and Control Register instruction has completed.

While not satisfying all of the conditions described above, \( \text{mffsce, mffscdrn}[i] \), \( \text{mffscrn}[i] \), and \( \text{mffsl} \) still obey the sequential execution model. Any FPSCR status bits read by \( \text{mffsce} \) or \( \text{mffsl} \) will reflect updates due to all preceding floating-point instructions. That is, all floating-point instructions following an \( \text{mffsce, mffscdrn}[i] \), or \( \text{mffscrn}[i] \) will execute based on any updates applied to any control bits in the FPSCR by the \( \text{mffsce, mffscdrn}[i] \), or \( \text{mffscrn}[i] \).

(Floating-point Storage Access instructions are not affected.)

The instruction descriptions in this section refer to “FPSCR fields,” where FPSCR field \( k \) is FPSCR bits \( 4k:4k+3 \).

### Move From FPSCR X-form

**mffs FRT**

\( \text{(Rc=0)} \)

**mffs. FRT**

\( \text{(Rc=1)} \)

<table>
<thead>
<tr>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>//</th>
<th>583</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>//</td>
<td>21</td>
<td>583</td>
</tr>
</tbody>
</table>

The contents of the FPSCR are placed into register FRT.

If \( \text{Rc=1} \), CR field 1 is set to the value \( FX||FE||WX||OX \).

**Special Registers Altered:**

\( \text{CR1} \)

\( \text{(if Rc=1)} \)

### Move From FPSCR & Clear Enables X-form

**mffsce FRT**

The contents of bits 56:60 (\( \text{VE, OE, UE, ZE, XE} \)) of the FPSCR are set to 0.

**Special Registers Altered:**

\( \text{VE (set to 0) OE (set to 0) UE (set to 0) ZE (set to 0) XE (set to 0)} \)

### Move From FPSCR Control & Set DRN X-form

**mffscdrn FRT,FRB**

Let \( \text{new}_{-}\text{DRN} \) be the contents of bits 29:31 of register FRB.

The contents of the control bits in the FPSCR, that is, bits 29:31 (\( \text{DRN} \)) and bits 56:63 (\( \text{VE, OE, UE, ZE, XE, NI, RN} \)), are placed into the corresponding bits in register FRT. All other bits in register FRT are set to 0.

\( \text{new}_{-}\text{DRN} \) is placed into bits 62:64 of the FPSCR (\( \text{DRN} \)).

**Special Registers Altered:**

\( \text{DRN} \)

**Programming Note**

\( \text{mffscdrn} \) permits software to simultaneously read control bits in the FPSCR and set the DRN field without the higher latency typically associated with accessing the status bits.
Move From FPSCR Control & Set DRN Immediate X-form

\[
mffscdri \quad \text{FRT,DRM}
\]

<table>
<thead>
<tr>
<th>63</th>
<th>FRT</th>
<th>21</th>
<th>DRN</th>
<th>583</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>18</td>
<td>21</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

The contents of the control bits in the FPSCR, that is, bits 29:31 (DRN) and bits 56:63 (VE, OE, UE, ZE, XE, NI, RN), are placed into the corresponding bits in register FRT. All other bits in register FRT are set to 0.

The contents of bits 29:31 of the FPSCR (DRN) are set to the value of DRM.

**Special Registers Altered:**

DRN

--- Programming Note ---

\[mffscdri\] permits software to simultaneously read control bits in the FPSCR and set the DRN field without the higher latency typically associated with accessing the status bits.

**Move From FPSCR Control & Set RN X-form**

\[
mffscrn \quad \text{FRT,FRB}
\]

<table>
<thead>
<tr>
<th>63</th>
<th>FRT</th>
<th>22</th>
<th>FRB</th>
<th>583</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

Let new_RN be the contents of bits 62:63 of register FRB.

The contents of the control bits in the FPSCR, that is, bits 29:31 (DRN) and bits 56:63 (VE, OE, UE, ZE, XE, NI, RN), are placed into the corresponding bits in register FRT. All other bits in register FRT are set to 0.

new_RN is placed into bits 62:63 of the FPSCR (RN).

**Special Registers Altered:**

RN

--- Programming Note ---

\[mffscrn\] permits software to simultaneously read control bits in the FPSCR and set the RN field without the higher latency typically associated with accessing the status bits.

--- Programming Note ---

\[mffscrn\] permits software to simultaneously read control bits in the FPSCR and set the RN field without the higher latency typically associated with accessing the status bits.

**Move From FPSCR Lightweight X-form**

\[
mffsl \quad \text{FRT}
\]

<table>
<thead>
<tr>
<th>63</th>
<th>FRT</th>
<th>24</th>
<th>583</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

The contents of the control bits in the FPSCR, that is, bits 29:31 (DRN) and bits 56:63 (VE, OE, UE, ZE, XE, NI, RN), and the non-sticky status bits in the FPSCR, that is, bits 45:51 (FR, FI, C, FL, FG, FE, FU), are placed into the corresponding bits in register FRT. All other bits in register FRT are set to 0.

**Special Registers Altered:**

None

--- Programming Note ---

\[mffsl\] permits software to read the control and non-sticky status bits in the FPSCR without the higher latency typically associated with accessing the sticky status bits.
Move to Condition Register from FPSCR

X-form

mcrfs BF,BFA

<table>
<thead>
<tr>
<th>63</th>
<th>BF // BFA // // /// 64 //</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6  9 11 14 16 21 31</td>
</tr>
</tbody>
</table>

The contents of FPSCR32:63 field BFA are copied to Condition Register field BF. All exception bits copied are set to 0 in the FPSCR. If the FX bit is copied, it is set to 0 in the FPSCR.

Special Registers Altered:

CR field

BF FX (if BFA=0)
UX ZX XX VXSAN (if BFA=1)
VXSI VXDI VXZDZ VXMZ (if BFA=2)
VXVC (if BFA=3)
VXSOFT VXSOQT VXCVI (if BFA=5)

Move To FPSCR Field Immediate X-form

mtfsfi BF,U,W

<table>
<thead>
<tr>
<th>63</th>
<th>BF // /// W U // 134 //</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6  9 11 14 16 21 22 31</td>
</tr>
</tbody>
</table>

The value of the U field is placed into FPSCR field BF+8*(1-W).

FX is altered only if BF=0 and W=0.

Special Registers Altered:

FPSCR field BF + 8*(1-W)
CR1 (if Rc=1)

mtfsfi serves as both a basic and an extended mnemonic. The Assembler will recognize a mtfsfi mnemonic with three operands as the basic form, and a mtfsfi mnemonic with two operands as the extended form. In the extended form the W operand is omitted and assumed to be 0.

Programming Note

When FPSCR32:35 is specified, bits 32 (FX) and 35 (OX) are set to the values of U2 and U3 (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from U2 and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule, given on page 133, and not from U3:2.

Move To FPSCR Fields XFL-form

mtfsf FLM,FRB,L,W

<table>
<thead>
<tr>
<th>63</th>
<th>L</th>
<th>FLM // 134 // FRB 711 //</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>7 11 14 16 21 22 31</td>
</tr>
</tbody>
</table>

The FPSCR is modified as specified by the FLM, L, and W fields.

L=0

The contents of register FRB are placed into the FPSCR under control of the W field and the field mask specified by FLM. W and the field mask identify the 4-bit fields affected. Let i be an integer in the range 0-7. If FLM_i=1 then FPSCR field k is set to the contents of the corresponding field of register FRB, where k=i+8*(1-W).

L=1

The contents of register FRB are placed into the FPSCR.

FX is not altered implicitly by this instruction.

Special Registers Altered:

FPSCR fields selected by mask, L, and W CR1 (if Rc=1)

Programming Note

mtfsf serves as both a basic and an extended mnemonic. The Assembler will recognize a mtfsf mnemonic with four operands as the basic form, and a mtfsf mnemonic with two operands as the extended form. In the extended form the W and L operands are omitted and both are assumed to be 0.

Programming Note

If L=1 or if L=0 and FPSCR32:35 is specified, bits 32 (FX) and 35 (OX) are set to the values of (FRB)32 and (FRB)35 (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from (FRB)32 and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule, given on page 133, and not from (FRB)33:34.
Move To FPSCR Bit 0 X-form

\begin{align*}
\text{mtfsb0} & \quad \text{BT} \quad (Rc=0) \\
\text{mtfsb0} \cdot & \quad \text{BT} \quad (Rc=1)
\end{align*}

Bit BT+32 of the FPSCR is set to 0.

Special Registers Altered:
- FPSCR bit BT+32
- CR1 \quad (if Rc=1)

\begin{table}
\begin{tabular}{cccccc}
63 & 6 & 11 & 16 & 21 & 31 \\
0 & & & & &
\end{tabular}
\end{table}

Programming Note

Bits 33 and 34 (FEX and VX) cannot be explicitly reset.

Move To FPSCR Bit 1 X-form

\begin{align*}
\text{mtfsb1} & \quad \text{BT} \quad (Rc=0) \\
\text{mtfsb1} \cdot & \quad \text{BT} \quad (Rc=1)
\end{align*}

Bit BT+32 of the FPSCR is set to 1.

Special Registers Altered:
- FPSCR bits BT+32 and FX
- CR1 \quad (if Rc=1)

\begin{table}
\begin{tabular}{cccccc}
63 & 6 & 11 & 16 & 21 & 31 \\
0 & & & & &
\end{tabular}
\end{table}

Programming Note

Bits 33 and 34 (FEX and VX) cannot be explicitly set.
Chapter 5. Decimal Floating-Point

5.1 Decimal Floating-Point (DFP) Facility Overview

This chapter describes the behavior of the decimal floating-point facility, the supported data types, formats, and classes, and the usage of registers. Also included are the execution model, exceptions, and instructions supported by the decimal floating-point facility.

The decimal floating-point (DFP) facility shares the 32 floating-point registers (FPRs) and the Floating-Point Status and Control Register (FPSCR) with the floating-point (BFP) facility. However, the interpretation of data formats in the FPRs, and the meaning of some control and status bits in the FPSCR are different between the BFP and DFP facilities.

The DFP facility also shares the Condition Register (CR) with the fixed-Point facility, the BFP facility, and the vector facility.

The DFP facility supports three DFP data formats: DFP Short (single precision), DFP Long (double precision), and DFP Extended (quad precision). Most operations are performed on DFP Long or DFP Extended format directly. Support for DFP Short is limited to conversion to and from DFP Long. Some DFP instructions operate on other data types, including signed or unsigned binary fixed-point data, and signed or unsigned decimal data.

DFP instructions are provided to perform arithmetic, compare, test, quantum-adjustment, conversion, and format operations on operands held in FPRs or FPR pairs.

- Arithmetic instructions
  These instructions perform addition, subtraction, multiplication, and division operations.

- Compare instructions
  These instructions perform a comparison operation on the numerical value of two DFP operands.

- Test instructions
  These instructions test the data class, the data group, the exponent, or the number of significant digits of a DFP operand.

- Quantum-adjustment instructions
  These instructions convert a DFP number to a result in the form that has the designated exponent, which may be explicitly or implicitly specified.

- Conversion instructions
  These instructions perform conversion between different data formats or data types.

- Format instructions
  These instructions facilitate composing or decomposing a DFP operand.

These instructions are described in Section 5.6 “DFP Instruction Descriptions” on page 205.

The three DFP data formats allow finite numbers to be represented with different precision and ranges. Special codes are also provided to represent +Infinity, -Infinity, Quiet NaN (Not-a-Number), and Signaling NaN. Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. The encoding of NaNs provides a diagnostic information field. This diagnostic field may be used to indicate such things as the source of an uninitialized variable or the reason an invalid result was produced.

The DFP processor recognizes a set of DFP exceptions which are indicated via bits set in the FPSCR. Additionally, the DFP exception actions depend on the setting of the various exception enable bits in the FPSCR.
The following DFP exceptions are detected by the DFP processor. The exception status bits in the FPSCR are indicated in parentheses.

- Invalid Operation Exception (VX)
  - SNaN (VXSNAN)
  - ∞ - ∞ (VXI SI)
  - ∞ ÷ ∞ (VXI DI)
  - 0 ÷ 0 (VXZDZ)
  - ∞ × 0 (VXIMZ)
  - Invalid Compare (VXVC)
  - Invalid conversion (VXCVI)

- Zero Divide Exception (ZX)

- Overflow Exception (OX)

- Underflow Exception (UX)

- Inexact Exception (XX)

Each DFP exception and each category of Invalid Operation Exception has an exception status bit in the FPSCR. In addition, each of the five DFP exceptions has a corresponding enable bit in the FPSCR. These enable bits enable or disable the invocation of the system floating-point enabled exception error handler, and may affect the setting of some exception status bits in the FPSCR.

The usage of these bits by the DFP facility differs from the usage by the BFP facility. Section 5.5.10 “DFP Exceptions” on page 197 provides a detailed discussion of DFP exceptions, including the effects of the enable bits.

### 5.2 DFP Register Handling

The following sections describe first how the floating-point registers are utilized by the DFP facility. The subsequent section covers the DFP usage of CR and FPSCR.

#### 5.2.1 DFP Usage of Floating-Point Registers

The DFP facility shares the same 32 64-bit FPRs with the BFP facility. Like the FP instructions, DFP instructions also use 5-bit fields for designating the FPRs to hold the source or target operands.

When data in DFP Short format is held in a FPR, it occupies the rightmost 32 bits of the FPR. The Load Floating-Point as Integer Word Algebraic instruction is provided to load the rightmost 32 bits of a FPR with a single-word data from storage. The Store Floating-Point as Integer Word instruction is available to store the rightmost 32 bits of a FPR to a storage location.

Data in DFP Long format, 64-bit binary fixed-point values, or 64-bit BCD values is held in a FPR using all 64 bits. Data of 64 bits may be loaded from storage via any of the Load Floating-Point Double instructions and stored via any of the Store Floating-Point Double instructions.

Data in DFP Extended format or 128-bit BCD values is held in an even-odd FPR pair using all 128 bits. Data of 128 bits must be loaded into the desired even-odd pair of floating-point registers using an appropriate sequence of the Load Floating-Point Double instructions and stored using an appropriate sequence of the Store Floating-Point Double instructions.

Data used as a source operand by any Decimal Floating-Point instruction that was produced, either directly or indirectly, by a Load Floating-Point Single instruction, a Floating Round to Single-Precision instruction, or a binary floating-point single-precision arithmetic instruction is boundedly undefined.

When an even-odd FPR pair is used to hold a 128-bit operand, the even-numbered FPR is used to hold the leftmost doubleword of the operand and the next higher-numbered FPR is used to hold the rightmost doubleword. A DFP instruction designating an odd-numbered FPR for a 128-bit operand is an invalid instruction form.

### Bit Definitions

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:28</td>
<td>Reserved</td>
</tr>
<tr>
<td>29:31</td>
<td>DFP Rounding Control (DRN)</td>
</tr>
<tr>
<td>32</td>
<td>Floating-Point Exception Summary (FX)</td>
</tr>
</tbody>
</table>

### Programming Note

The Floating-Point Move instructions can be used to move operands between FPRs.

The bit definitions for the FPSCR are as follows.

#### Programming Note

FPSCR28 is reserved for extension of the DRN field, therefore DRN may be set using the mtfsfi instruction to set the rounding mode.

Floating-Point Exception Summary (FX)

Every floating-point instruction, except mtfsi and mtfs, implicitly sets FX to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1.
**Floating-Point Fraction Rounded (FF)**
The last Arithmetic or Rounding and Conversion instruction incremented the fraction during rounding. See Section 5.5.1, “Rounding” on page 194. This bit is not sticky.

**Floating-Point Inexact (FI)**
The last Arithmetic or Rounding and Conversion instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 5.5.1. This bit is not sticky.

See the definition of XX, above, regarding the relationship between FI and XX.

**Floating-Point Result Flags (FPRF)**
This field is set as described below. For arithmetic, rounding, and conversion instructions, the field is set based on the result placed into the target register, except that if any portion of the result is undefined then the value placed into FPRF is undefined.

**Floating-Point Result Class Descriptor (C)**
Arithmetic, rounding, and conversion instructions may set this bit with the FPRC bits, to indicate the class of the result as shown in Figure 58 on page 190.

**Floating-Point Condition Code (FPCC)**
Floating-point Compare and DFP Test instructions set one of the FPCC bits to 1 and the other three FPCC bits to 0. Arithmetic, rounding, and conversion instructions may set the FPCC bits with the C bit, to indicate the class of the result as shown in Figure 58 on page 190. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

**Floating-Point Less Than or Negative (FL or <)**

**Floating-Point Greater Than or Positive (FG or >)**

**Floating-Point Equal or Zero (FE or =)**

**Floating-Point Unordered or NaN (FU or ?)**
Reserved

**Floating-Point Invalid Operation Exception (Software Request) (VXSOFT)**
This bit can be altered only by **mcrfs, mtfsfi, mtfsf, mtfsb0, or mtfsb1**. See Section 5.5.10.1, “Invalid Operation Exception” on page 199.

Neither used nor changed by DFP.

---

Floating-Point Enabled Exception Summary (FEX)
This bit is the OR of all the floating-point exception bits masked by their respective enable bits. **mcrfs, mtfsfi, mtfsf, mtfsb0,** and **mtfsb1** cannot alter FEX explicitly.

Floating-Point Invalid Operation Exception Summary (XX)
This bit is the OR of all the Invalid Operation exception bits. **mcrfs, mtfsfi, mtfsf, mtfsb0,** and **mtfsb1** cannot alter XX explicitly.

Floating-Point Overflow Exception (OX)
See Section 5.5.10.3, “Overflow Exception” on page 201.

Floating-Point Underflow Exception (UX)
See Section 5.5.10.4, “Underflow Exception” on page 201.

Floating-Point Zero Divide Exception (ZX)
See Section 5.5.10.2, “Zero Divide Exception” on page 200.

Floating-Point Inexact Exception (XX)
See Section 5.5.10.5, “Inexact Exception” on page 202.

XX is a sticky version of FI (see below). Thus the following rules completely describe how XX is set by a given instruction:

- If the instruction affects FI, the new value of XX is obtained by ORing the old value of XX with the new value of FI.
- If the instruction does not affect FI, the value of XX is unchanged.

Floating-Point Invalid Operation Exception (VXSNAN)
See Section 5.5.10.1, “Invalid Operation Exception” on page 199.

Floating-Point Invalid Operation Exception (Infinity – Infinity) (VXI SI)
See Section 5.5.10.1.

Floating-Point Invalid Operation Exception (Infinity > Infinity) (VXI DI)
See Section 5.5.10.1.

Floating-Point Invalid Operation Exception (Zero > Zero) (VX2ZD)
See Section 5.5.10.1.

Floating-Point Invalid Operation Exception (Infinity × Zero) (VX1 MZ)
See Section 5.5.10.1.

Floating-Point Invalid Operation Exception (Invalid Compare) (VXVC)
See Section 5.5.10.1.

Floating-Point Enabled Exception Summary (FEX)
This bit is the OR of all the floating-point exception bits masked by their respective enable bits. **mcrfs, mtfsfi, mtfsf, mtfsb0,** and **mtfsb1** cannot alter FEX explicitly.

Floating-Point Overflow Exception
This bit is the OR of all the Invalid Operation exception bits. **mcrfs, mtfsfi, mtfsf, mtfsb0,** and **mtfsb1** cannot alter XX explicitly.

Floating-Point Inexact Exception (XX)
See Section 5.5.10.5, “Inexact Exception” on page 202.

XX is a sticky version of FI (see below). Thus the following rules completely describe how XX is set by a given instruction:

- If the instruction affects FI, the new value of XX is obtained by ORing the old value of XX with the new value of FI.
- If the instruction does not affect FI, the value of XX is unchanged.

Floating-Point Invalid Operation Exception (VXSNAN)
See Section 5.5.10.1, “Invalid Operation Exception” on page 199.
5.3 DFP Support for Non-DFP Data Types

In addition to the DFP data types, the DFP processor provides limited support for the following non-DFP data types: signed or unsigned binary fixed-point data, and signed or unsigned decimal data.

In unsigned binary fixed-point data, all bits are used to express the absolute value of the number. For signed binary fixed-point data, the leftmost bit represents the sign, which is followed by the numeric field. Positive numbers are represented in true binary notation with the sign bit set to zero. When the value is zero, all bits are zeros, including the sign bit. Negative numbers are represented in two's complement binary notation with a one in the sign-bit position.

For decimal data, each byte contains a pair of four-bit nibbles; each four-bit nibble contains a binary-coded-decimal (BCD) code. There are two kinds of BCD codes: digit code and sign code. For unsigned decimal data, all nibbles contain a digit code as shown in Figure 59.

![Figure 59. Format for Unsigned Decimal Data](image_url)

For signed decimal data, the rightmost nibble contains a sign code (S) and all other nibbles contain a digit code as shown in Figure 60.

![Figure 60. Format for Signed Decimal Data](image_url)

The decimal digits 0-9 have the binary encoding 0000-1001. The preferred plus-sign codes are 1100 and 1111. The preferred minus sign code is 1101. These are the sign codes generated for the results of the Decode DPD To BCD instruction. A selection is provided by this instruction to specify which of the two preferred plus sign codes is to be generated. Alternate sign codes are also recognized as valid in the sign position: 1010 and 1011 are alternate sign codes for plus, and 1011 is an alternate sign code for minus. Alternate sign codes are accepted for any source operand, but are not generated as a result by the instruction. When an invalid digit or sign code is detected by the Encode BCD To DPD instruction, an invalid-operation exception occurs. A
5.4 DFP Number Representation

A DFP finite number consists of three components: a sign bit, a signed exponent, and a significand. The signed exponent is a signed binary integer. The significand consists of a number of decimal digits, which are to the left of the implied decimal point. The rightmost digit of the significand is called the units digit. The numerical value of a DFP finite number is represented as \((-1)^{\text{sign}} \times \text{significand} \times 10^{\text{exponent}}\) and the unit value of this number is \((1 \times 10^{\text{exponent}})\), which is called the quantum.

DFP finite numbers are not normalized. This allows leading zeros and trailing zeros to exist in the significand. This unnormalized DFP number representation allows some values to have redundant forms; each form represents the DFP number with a different combination of the significand value and the exponent value. For example, \(1000000 \times 10^5\) and \(10 \times 10^{10}\) are two different forms of the same numerical value. A form of this number representation carries information about both the numerical value and the quantum of a DFP finite number.

The significant digits of a DFP finite number are the digits in the significand beginning with the leftmost non-zero digit and ending with the units digit.

### 5.4.1 DFP Data Format

DFP numbers and NaNs may be represented in FPRs in any of the three data formats: DFP Short, DFP Long, or DFP Extended. The contents of each data format represent encoded information. Special codes are assigned to NaNs and infinities. Different formats support different sizes in both significand and exponent. Arithmetic, compare, test, quantum-adjustment, and format instructions are provided for DFP Long and DFP Extended formats only.

The sign is encoded as a one bit binary value. Significand is encoded as an unsigned decimal integer in two distinct parts. The leftmost digit (LMD) of the significand is encoded as part of the combination field; the remaining digits of the significand are encoded in the trailing significand field. The exponent is contained in the combination field in two parts. However, prior to encoding, the exponent is converted to an unsigned binary value called the biased exponent by adding a bias value which is a constant for each format. The two leftmost bits of the biased exponent are encoded with the leftmost digit of the significand in the leftmost bits of the combination field. The rest of the biased exponent occupies the remaining portion of the combination field.

### 5.4.1.1 Fields Within the Data Format

The DFP data representation comprises three fields, as diagrammed below for each of the three formats:

#### DFP Short format

- **Sign bit (S)**
- **Combination field (G)**
- **Trailing significand (T)**

#### DFP Long format

- **Sign bit (S)**
- **Combination field (G)**
- **Trailing significand (T)**

#### DFP Extended format

- **Sign bit (S)**
- **Combination field (G)**
- **Trailing significand (T)**

The fields are defined as follows:

- **Sign bit (S)**
  
The sign bit is in bit 0 of each format, and is zero for plus and one for minus.

- **Combination field (G)**
  
  As the name implies, this field provides a combination of the exponent and the left-most digit (LMD) of the significand, for finite numbers, or provides a special code.
for denoting the value as either a Not-a-Number or an Infinity.

The first 5 bits of the combination field contain the encoding of NaN or infinity, or the two leftmost bits of the biased exponent and the leftmost digit (LMD) of the significand. The following tables show the encoding:

<table>
<thead>
<tr>
<th>G₀:₄</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111</td>
<td>NaN</td>
</tr>
<tr>
<td>11110</td>
<td>Infinity</td>
</tr>
<tr>
<td>All others</td>
<td>Finite Number (see Figure 66)</td>
</tr>
</tbody>
</table>

Figure 65. Encoding of the G field for Special Symbols

<table>
<thead>
<tr>
<th>LMD</th>
<th>Leftmost 2-bits of biased exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>00000</td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
</tr>
<tr>
<td>2</td>
<td>00010</td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
</tr>
<tr>
<td>8</td>
<td>11000</td>
</tr>
<tr>
<td>9</td>
<td>11001</td>
</tr>
</tbody>
</table>

Figure 66. Encoding of bits 0:4 of the G field for Finite Numbers

For DFP finite numbers, the rightmost N-5 bits of the N-bit combination field contain the remaining bits of the biased exponent. For NaNs, bit 5 of the combination field is used to distinguish a Quiet NaN from a Signaling NaN; the remaining bits in a source operand are ignored and they are set to zeros in a target operand by most operations. For infinities, the rightmost N-5 bits of the N-bit combination field of a source operand are ignored and they are set to zeros in a target operand by most operations.

**Trailing Significand field (T)**

For DFP finite numbers, this field contains the remaining *(significant digits). For NaNs, this field may be used to contain diagnostic information. For infinities, contents in this field of a source operand are ignored and they are set to zeros in a target operand by most operations. The trailing significand field is a multiple of 10-bit blocks. The multiple depends on the format. Each 10-bit block is called a declet and represents three decimal digits, using the Densely Packed Decimal (DPD) encoding defined in Appendix B.

### 5.4.1.2 Summary of DFP Data Formats

The properties of the three DFP formats are summarized in the following table:

<table>
<thead>
<tr>
<th>Widths (bits):</th>
<th>Format</th>
<th>Sign (S)</th>
<th>Combination (G)</th>
<th>Trailing Significand (T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFP Short</td>
<td>32</td>
<td>1</td>
<td>11</td>
<td>20</td>
</tr>
<tr>
<td>DFP Long</td>
<td>64</td>
<td>1</td>
<td>13</td>
<td>50</td>
</tr>
<tr>
<td>DFP Extended</td>
<td>128</td>
<td>1</td>
<td>17</td>
<td>110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exponent:</th>
<th>Maximum biased</th>
<th>Maximum (Xₘₜₜ)</th>
<th>Minimum (Xₘₜₜ)</th>
<th>Bias</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFP Short</td>
<td>191</td>
<td>90</td>
<td>-101</td>
<td>101</td>
</tr>
<tr>
<td>DFP Long</td>
<td>767</td>
<td>369</td>
<td>-398</td>
<td>398</td>
</tr>
<tr>
<td>DFP Extended</td>
<td>12,287</td>
<td>6111</td>
<td>-6176</td>
<td>6176</td>
</tr>
</tbody>
</table>

| Precision (p) (digits) | 7 | 16 | 34 |

<table>
<thead>
<tr>
<th>Magnitude:</th>
<th>Maximum normal number (Nₘₜₜ)</th>
<th>Minimum normal number (Nₘᵣₜ)</th>
<th>Minimum subnormal number (Oₘᵣₜ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFP Short</td>
<td>(10⁷ × 1) × 10⁹³</td>
<td>1 × 10⁻⁹⁵</td>
<td>1 × 10⁻¹⁰¹</td>
</tr>
<tr>
<td>DFP Long</td>
<td>(10¹⁶ × 1) × 10³⁶⁹</td>
<td>1 × 10⁻³⁸³</td>
<td>1 × 10⁻³⁹⁸</td>
</tr>
<tr>
<td>DFP Extended</td>
<td>(10³⁴ × 1) × 10¹¹¹</td>
<td>1 × 10⁻¹⁴³</td>
<td>1 × 10⁻¹⁷⁶</td>
</tr>
</tbody>
</table>

Figure 67. Summary of DFP Formats
5.4.1.3 Preferred DPD Encoding

Execution of DFP instructions decodes source operands from DFP data formats to an internal format for processing, and encodes the operation result before the final result is returned as the target operand.

As part of the decoding process, declets in the trailing significand field of source operands are decoded to their corresponding BCD digit codes using the DPD-to-BCD decoding algorithm. As part of the encoding process, BCD digit codes to be stored into the trailing significand field of the target operand are encoded into declets using the BCD-to-DPD encoding algorithm. Both the decoding and encoding algorithms are defined in Appendix B.

As explained in Appendix B, there are eight 3-digit decimal values that have redundant DPD codes and one preferred DPD code. All redundant DPD codes are recognized in source operands for the associated 3-digit decimal number. DFP operations will always generate the preferred DPD codes for the trailing significand field of the target operand.

5.4.2 Classes of DFP Data

There are six classes of DFP data, which include numerical and nonnumeric entities. The numerical entities include zero, subnormal number, normal number, and infinity data classes. The nonnumeric entities include quiet and signaling NaNs data classes. The value of a DFP finite number, including zero, subnormal number, and normal number, is a quantization of the real number based on the data format. The Test Data Class instruction may be used to determine the class of a DFP operand. In general, an operation that returns a DFP result sets the FPRF field to indicate the data class of the result.

The following tables show the value ranges for finite-number data classes, and the codes for NaNs and infinities.

<table>
<thead>
<tr>
<th>Data Class</th>
<th>Sign</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>±</td>
<td>0*</td>
</tr>
<tr>
<td>Subnormal</td>
<td>±</td>
<td>$D_{\text{min}} \leq</td>
</tr>
<tr>
<td>Normal</td>
<td>±</td>
<td>$N_{\text{min}} \leq</td>
</tr>
</tbody>
</table>

* The significand is zero and the exponent is any representable value

![Figure 68. Value Ranges for Finite Number Data Classes](image)

<table>
<thead>
<tr>
<th>Data Class</th>
<th>S</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Infinity</td>
<td>0</td>
<td>11110xxx . . . xxx</td>
<td>xxx . . . xxx</td>
</tr>
<tr>
<td>-Infinity</td>
<td>1</td>
<td>11110xxx . . . xxx</td>
<td>xxx . . . xxx</td>
</tr>
<tr>
<td>Quiet NaN</td>
<td>x</td>
<td>11110xxx . . . xxx</td>
<td>xxx . . . xxx</td>
</tr>
<tr>
<td>Signaling NaN</td>
<td>x</td>
<td>111110xxx . . . xxx</td>
<td>xxx . . . xxx</td>
</tr>
<tr>
<td>x Don't care</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 69. Encoding of NaN and Infinity Data Classes](image)

**Zeros**

Zeros have a zero significand and any representable value in the exponent. A +0 is distinct from -0, and zeros with different exponents are distinct, except that comparison treats them as equal.

**Subnormal Numbers**

Subnormal numbers have values that are smaller than $N_{\text{min}}$, and greater than zero in magnitude.

**Normal Numbers**

Normal numbers are nonzero finite numbers whose magnitude is between $N_{\text{min}}$ and $N_{\text{max}}$ inclusively.

**Infinities**

Infinities are represented by $0b11110$ in the leftmost 5 bits of the combination field. When an operation is defined to generate an infinity as the result, a default infinity is sometimes supplied. A default infinity has all remaining bits in the combination field and trailing significand field set to zeros.

When infinities are used as source operands, only the leftmost 5 bits of the combination field are interpreted (i.e., $0b11110$ indicates the value is an infinity). The trailing significand field of infinities is usually ignored. For generated infinities, the leftmost 5 bits of the combination field are set to $0b11110$ and all remaining combination bits are set to zero.

Infinities can participate in most arithmetic operations and give a consistent result. In comparisons, any +Infinity compares greater than any finite number, and any -Infinity compares less than any finite number. All +Infinity are compared equal and all -Infinity are compared equal.

**Signaling and Quiet NaNs**

There are two types of Not-a-Numbers (NaNs), Signaling (SNaN) and Quiet (QNaN).

$0b111110$ in the leftmost 6 bits of the combination field indicates a Quiet NaN, whereas $0b111111$ indicates a Signaling NaN.

A special QNaN is sometimes supplied as the default QNaN for a disabled invalid-operation exception; it has a plus sign, the leftmost 6 bits of the combination field set to $0b111110$ and remaining bits in the combination field and the trailing significand field set to zero.

Normally, source QNaNs are propagated during operations so that they will remain visible at the end. When a
QNaN is propagated, the sign is preserved, the decimal value of the trailing significand field is preserved but reencoded using the preferred DPD codes, and the contents in the rightmost \( n - 6 \) bits of the combination field set to zero, where \( n \) is the width of the combination field for the format.

A source SNaN generally causes an invalid-operation exception. If the exception is disabled, the SNaN is converted to the corresponding QNaN and propagated. The primary encoding difference between an SNaN and a QNaN is that bit 5 of an SNaN is 1 and bit 5 of a QNaN is 0. When an SNaN is propagated as a QNaN, bit 5 is set to 0, and, just as with QNaN propagation, the sign is preserved, the decimal value of the trailing significand field is preserved but reencoded using the preferred DPD codes, and the contents in the rightmost \( n - 6 \) bits of the combination field set to zero, where \( n \) is the width of the combination field for the format. For some format-conversion instructions, a source SNaN does not cause an invalid-operation exception, and an SNaN is returned as the target operand.

For instructions with two source NaNs and a NaN is to be propagated as the result, do the following.
- If there is a QNaN in \( F_R \) and an SNaN in \( F_B \), the SNaN in \( F_R \) is propagated.
- Otherwise, propagate the NaN is \( F_R \).

5.5 DFP Execution Model

DFP operations are performed as if they first produce an intermediate result correct to infinite precision and with unbounded range. The intermediate result is then rounded to the destination’s precision according to one of the eight DFP rounding modes. If the rounded result has only one form, it is delivered as the final result; if the rounded result has redundant forms, then an ideal exponent is used to select the form of the final result. The ideal exponent determines the form, not the value, of the final result. (See Section 5.5.3 “Formation of Final Result” on page 195.)

5.5.1 Rounding

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit the destination’s precision. The destination’s precision of an operation defines the set of permissible resultant values. For most operations, the destination’s precision is the target-format precision and the permissible resultant values are those values representable in the target format. For some special operations, the destination precision is constrained by both the target format and some additional restrictions, and the permissible resultant values are a subset of the values representable in the target format.

Rounding sets FPSCR bits \( F_R \) and \( F_I \). When an inexact exception occurs, \( F_I \) is set to one; otherwise, \( F_I \) is set to zero. When an inexact exception occurs and if the rounded result is greater in magnitude than the intermediate result, then \( F_R \) is set to one; otherwise, \( F_R \) is set to zero. The exception is the Round to FP Integer Without Inexact instruction, which always sets \( F_R \) and \( F_I \) to zero. Rounding may cause an overflow exception or underflow exception; it may also cause an inexact exception.

Refer to Figure 70 below for rounding. Let \( Z \) be the intermediate result of a DFP operation. \( Z \) may or may not fit in the destination’s precision. If \( Z \) is exactly one of the permissible representable resultant values, then the final result in all rounding modes is \( Z \). Otherwise, either \( Z_1 \) or \( Z_2 \) is chosen to approximate the result, where \( Z_1 \) and \( Z_2 \) are the next larger and smaller permissible resultant values, respectively.

![Figure 70. Rounding](image)

**Round to Nearest, Ties to Even**

Choose the value that is closer to \( Z \) (\( Z_1 \) or \( Z_2 \)). In case of a tie, choose the one whose units digit would have been even in the form with the largest common quantum of the two permissible resultant values. However, an infinitely precise result with magnitude at least \( (N_{max} + 0.5Q(N_{max})) \) is rounded to infinity with no change in sign; where \( Q(N_{max}) \) is the quantum of \( N_{max} \).

**Round toward 0**

Choose the smaller in magnitude (\( Z_1 \) or \( Z_2 \)).

**Round toward \( +\infty \)**

Choose \( Z_1 \).

**Round toward \( -\infty \)**

Choose \( Z_2 \).

**Round to Nearest, Ties away from 0**

Choose the value that is closer to \( Z \) (\( Z_1 \) or \( Z_2 \)). In case of a tie, choose the larger in magnitude (\( Z_1 \) or \( Z_2 \)). However, an infinitely precise result with magnitude at least \( (N_{max} + 0.5Q(N_{max})) \) is rounded to infinity with no change in sign; where \( Q(N_{max}) \) is the quantum of \( N_{max} \).

**Round to Nearest, Ties toward 0**

Choose the value that is closer to \( Z \) (\( Z_1 \) or \( Z_2 \)). In case of a tie, choose the smaller in magnitude (\( Z_1 \) or \( Z_2 \)). However, an infinitely precise result with magnitude greater than \( (N_{max} + 0.5Q(N_{max})) \) is rounded to infinity.
with no change in sign; where $Q(N_{max})$ is the quantum of $N_{max}$.

**Round away from 0**
Choose the larger in magnitude ($Z_1$ or $Z_2$).

**Round to prepare for shorter precision**
Choose the smaller in magnitude ($Z_1$ or $Z_2$). If the selected value is inexact and the units digit of the selected value is either 0 or 5, then the digit is incremented by one and the incremented result is delivered. In all other cases, the selected value is delivered. When a value has redundant forms, the units digit is determined by using the form that has the smallest exponent.

### 5.5.2 Rounding Mode Specification

Unless otherwise specified in the instruction definition, the rounding mode used by an operation is specified in the DFP rounding control (DRN) field of the FPSCR. The eight DFP rounding modes are encoded in the DRN field as specified in the table below.

<table>
<thead>
<tr>
<th>DRN</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Round to Nearest, Ties to Even</td>
</tr>
<tr>
<td>001</td>
<td>Round toward 0</td>
</tr>
<tr>
<td>010</td>
<td>Round toward +Infinity</td>
</tr>
<tr>
<td>011</td>
<td>Round toward -Infinity</td>
</tr>
<tr>
<td>100</td>
<td>Round to Nearest, Ties away from 0</td>
</tr>
<tr>
<td>101</td>
<td>Round to Nearest, Ties toward 0</td>
</tr>
<tr>
<td>110</td>
<td>Round away from 0</td>
</tr>
<tr>
<td>111</td>
<td>Round to Prepare for Shorter Precision</td>
</tr>
</tbody>
</table>

**Figure 71. Encoding of DFP Rounding-Mode Control (DRN)**

For the quantum-adjustment, a 2-bit immediate field, called **RMC** (*Rounding Mode Control*), in the instruction specifies the rounding mode used. The RMC field may contain a primary encoding or a secondary encoding. For **Quantize**, **Quantize Immediate**, and **Reround**, the RMC field contains the primary encoding. For **Round to FP Integer** the field contains either encoding, depending on the setting of a RMC-encoding-selection bit. The following tables define the primary encoding and the secondary encoding.

<table>
<thead>
<tr>
<th>Primary RMC</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Round to nearest, ties to even</td>
</tr>
<tr>
<td>01</td>
<td>Round toward 0</td>
</tr>
<tr>
<td>10</td>
<td>Round to nearest, ties away from 0</td>
</tr>
<tr>
<td>11</td>
<td>Round according to DRN</td>
</tr>
</tbody>
</table>

**Figure 72. Primary Encoding of Rounding-Mode Control**

**Secondary RMC | Rounding Mode**

<table>
<thead>
<tr>
<th>RMC</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Round to +∞</td>
</tr>
<tr>
<td>01</td>
<td>Round to -∞</td>
</tr>
<tr>
<td>10</td>
<td>Round away from 0</td>
</tr>
<tr>
<td>11</td>
<td>Round to nearest, ties toward 0</td>
</tr>
</tbody>
</table>

**Figure 73. Secondary Encoding of Rounding-Mode Control**

### 5.5.3 Formation of Final Result

An ideal exponent is defined for each DFP instruction that returns a DFP data operand.

#### 5.5.3.1 Use of Ideal Exponent

For all DFP operations,
- if the rounded intermediate result has only one form, then that form is delivered as the final result.
- if the rounded intermediate result has redundant forms and is exact, then the form with the exponent closest to the ideal exponent is delivered.
- if the rounded intermediate result has redundant forms and is inexact, then the form with the smallest exponent is delivered.

The following table specifies the ideal exponent for each instruction.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Ideal Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>$\min(E(FRA), E(FRB))$</td>
</tr>
<tr>
<td>Subtract</td>
<td>$\min(E(FRA), E(FRB))$</td>
</tr>
<tr>
<td>Multiply</td>
<td>$E(FRA) + E(FRB)$</td>
</tr>
<tr>
<td>Divide</td>
<td>$E(FRA) - E(FRB)$</td>
</tr>
<tr>
<td>Quantize-Immediate</td>
<td>See Instruction Description</td>
</tr>
<tr>
<td>Quantize</td>
<td>$E(FRA)$</td>
</tr>
<tr>
<td>Reround</td>
<td>See Instruction Description</td>
</tr>
<tr>
<td>Round to FP Integer</td>
<td>$\max(0, E(FRA))$</td>
</tr>
<tr>
<td>Convert to DFP Long</td>
<td>$E(FRA)$</td>
</tr>
<tr>
<td>Convert to DFP</td>
<td>$E(FRA)$</td>
</tr>
<tr>
<td>Extended</td>
<td>$E(FRA)$</td>
</tr>
<tr>
<td>Round to DFP Short</td>
<td>$E(FRA)$</td>
</tr>
<tr>
<td>Round to DFP Long</td>
<td>$E(FRA)$</td>
</tr>
<tr>
<td>Convert from Fixed</td>
<td>0</td>
</tr>
<tr>
<td>Encode BCD to DPD</td>
<td>0</td>
</tr>
<tr>
<td>Insert Biased Exponent</td>
<td>$E(FRA)$</td>
</tr>
</tbody>
</table>

Notes:
- $E(x)$ - exponent of the DFP operand in register $x$.

**Figure 74. Summary of Ideal Exponents**
5.5.4 Arithmetic Operations

Four arithmetic operations are provided: Add, Subtract, Multiply, and Divide.

5.5.4.1 Sign of Arithmetic Result

The following rules govern the sign of an arithmetic operation when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the source operand having the larger absolute value. If both source operands have the same sign, the sign of the result of an add operation is the same as the sign of the source operands. When the sum of two operands with opposite signs is exactly zero, the sign of the result is positive in all rounding modes except Round toward -∞, in which case the sign is negative.
- The sign of the result of the subtract operation x - y is the same as the sign of the result of the add operation x + (-y).
- The sign of the result of a multiply or divide operation is the exclusive-OR of the signs of the source operands.

5.5.5 Compare Operations

Two sets of instructions are provided for comparing numerical values: Compare Ordered and Compare Unordered. In the absence of NaNs, these instructions work the same. These instructions work differently when either of the followings is true:

1. At least one source operand of the instruction is an SNaN and the invalid-operation exception is disabled.
2. When there is no SNaN in any source operand, at least one source operand of the instruction is a QNaN.

In case 1, Compare Unordered recognizes an invalid-operation exception and sets the VXSNAN flag, but Compare Ordered recognizes the exception and sets both the VXSNAN and VXVC flags. In case 2, Compare Unordered does not recognize an exception, but Compare Ordered recognizes an invalid-operation exception and sets the VXVC flag.

For finite numbers, comparisons are performed on values, that is, all redundant forms of a DFP number are treated equal.

Comparisons are always exact and cannot cause an inexact exception.

Comparison ignores the sign of zero, that is, +0 equals -0.

Infinities with like sign compare equal, that is, +∞ equals +∞, and -∞ equals -∞.

A NaN compares as unordered with any other operand, whether a finite number, an infinity, or another NaN, including itself.

Execution of a compare instruction always completes, regardless of whether any DFP exception occurs or not, and whether the exception is enabled or not.

5.5.6 Test Operations

Four kinds of test operations are provided: Test Data Class, Test Data Group, Test Exponent, and Test Significance.

The Test Data Class instruction examines the contents of a source operand and determines if the operand is one of the specified data classes. The test result and the sign of the source operand are indicated in the FPSCR field and CR field BF.

The Test Data Group instruction examines the contents of a source operand and determines if the operand is one of the specified data groups. The test result and the sign of the source operand are indicated in the FPCC field and CR field BF.

The Test Exponent instruction compares the exponent of the two source operands. The test operation ignores the sign and significand of operands. Infinities compare equal, and NaNs compare equal. The test result is indicated in the FPCC field and CR field BF.

Execution of a test instruction does not cause any DFP exception.

5.5.7 Quantum Adjustment Operations

Four kinds of quantum-adjustment operations are provided: Quantize, Quantize Immediate, Reround, and Round To FP Integer. Each of them has an immediate field which specifies whether the rounding mode in FPSCR or a different one is to be used.

The Quantize instruction is used to adjust a DFP number to the form that has the specified target exponent. The Quantize Immediate instruction is similar to the Quantize instruction, except that the target exponent is specified in a 5-bit immediate field as a signed binary integer and has a limited range.

The Reround instruction is used to simulate a DFP operation of a precision other than that of DFP Long or DFP Extended. For the Reround instruction to produce
a result which accurately reflects that which would have resulted from a DFP operation of the desired precision $d$ in the range $1:33$ inclusively, the following conditions must be met:

- The precision of the preceding DFP operation must be at least one digit larger than $d$.
- The rounding mode used by the preceding DFP operation must be round-to-prepare-for-shorter-precision.

The Round To FP Integer instruction is used to round a DFP number to an integer value of the same format. The target exponent is implicitly specified, and is greater than or equal to zero.

### 5.5.8 Conversion Operations

There are two kinds of conversion operations: data-format conversion and data-type conversion.

#### 5.5.8.1 Data-Format Conversion

The instructions Convert To DFP Long and Convert To DFP Extended convert DFP operands to wider formats; the instructions Round To DFP Short and Round To DFP Long convert DFP operands to narrower formats.

When converting a finite number to a wider format, the result is exact. When converting a finite number to a narrower format, the source operand is rounded to the target-format precision, which is specified by the instruction, not by the target register size.

When converting a finite number, the ideal exponent of the result is the source exponent.

Conversion of an infinity or NaN to a different format does not preserve the source combination field. Let $N$ be the width of the target format’s combination field.

- When the result is an infinity or a QNaN, the contents of the rightmost $N-5$ bits of the $N$-bit target combination field are set to zero.
- When the result is an SNaN, bit 5 of the target format’s combination field is set to one and the rightmost $N-6$ bits of the $N$-bit target combination field are set to zero.

When converting a NaN to a wider format or when converting an infinity from DFP Short to DFP Long, digits in the source trailing significand field are reencoded using the preferred DPD codes with sufficient zeros appended on the left to form the target trailing significand field. When converting a NaN to a narrower format or when converting an infinity from DFP Long to DFP Short, the appropriate number of leftmost digits of the source trailing significand field are removed and the remaining digits of the field are reencoded using the preferred DPD codes to form the target trailing significand field.

When converting an infinity between DFP Long and DFP Extended, a default infinity with the same sign is produced.

When converting an SNaN between DFP Short and DFP Long, it is converted to an SNaN without causing an invalid-operation exception. When converting an SNaN between DFP Long and DFP Extended, the invalid-operation exception occurs; if the invalid-operation exception is disabled, the result is converted to the corresponding QNaN.

#### 5.5.8.2 Data-Type Conversion

The instructions Convert From Fixed and Convert To Fixed are provided to convert a number between the DFP data type and the signed 64-bit binary-integer data type.

Conversion of a signed 64-bit binary integer to a DFP Extended number is always exact.

Conversion of a DFP number to a signed 64-bit binary integer results in an invalid-operation exception when the converted value does not fit into the target format, or when the source operand is an infinity or NaN. When the exception is disabled, the most positive integer is returned if the source operand is a positive number or $+\infty$, and the most negative integer is returned if the source operand is a negative number, $-\infty$, or NaN.

### 5.5.9 Format Operations

The format instructions are provided to facilitate composing or decomposing a DFP number, and consist of Encode BCD To DPD, Decode DPD To BCD, Extract Biased Exponent, Insert Biased Exponent, Shift Significand Left Immediate, and Shift Significand Right Immediate. A source operand of SNaN does not cause an invalid-operation exception, and an SNaN may be produced as the target operand.

### 5.5.10 DFP Exceptions

This architecture defines the following DFP exceptions:

- Invalid Operation Exception
  - SNaN
  - $\infty \times \infty$
  - $\infty \div \infty$
  - $0 \div 0$
  - Inexact Exception
  - Invalid Compare
  - Invalid Conversion
- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions may occur during execution of a DFP instruction.
Each DFP exception, and each category of the Invalid Operation Exception, has an exception status bit in the FPSCR. In addition, each DFP exception has a corresponding enable bit in the FPSCR. The exception status bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the fE0 and fE1 bits (see the discussion of fE0 and fE1 below), whether and how the system floating-point enabled exception error handler is invoked. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its source operands, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

A single instruction, other than mtfsfi or mtfsf, may set more than one exception bit only in the following cases:

- Inexact Exception may be set with Overflow Exception.
- Inexact Exception may be set with Underflow Exception.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Compare) for Compare Ordered instructions
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Conversion) for Convert To Fixed instructions.

When an exception occurs the instruction execution may be completed or partially completed, depending on the exception and the operation.

For all instructions, except for the Compare and Test instructions, the following exceptions cause the instruction execution to be partially completed. That is, setting of CR field 1 (when RC=1) and exception status flags is performed, but no result is stored into the target FPR or FPR pair. For Compare and Test instructions, instruction execution is always completed, regardless of whether any DFP exception occurs or not, and whether the exception is enabled or not.

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exceptions, instruction execution is completed, a result, if specified by the instruction, is generated and stored into the target FPR or FPR pair, and appropriate status flags are set. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exceptions that deliver a result in target FPR are the following:

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the DFP exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of “traps” and “trap handlers”. In this architecture, a FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the “trap enabled” case: the expectation is that the exception will be detected by software, which will revise the result. A FPSCR exception enable bit of 0 causes generation of the “default result” value specified for the “trap disabled” (or “no trap occurs” or “trap is not implemented”) case: the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to zero and Ignore Exceptions Mode (see below) should be used. In this case the system floating-point enabled exception error handler is not invoked, even if DFP exceptions occur: software can inspect the FPSCR exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to one and a mode other than Ignore Exceptions Mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled DFP exception occurs. The system floating-point enabled exception error handler is also invoked if a Move To FPSCR instruction causes an exception bit and the corresponding enable bit both to be 1; the Move To FPSCR instruction is considered to cause the enabled exception.

The fE0 and fE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled DFP exception occurs. The location of these bits and the requirements for altering them are described in Book III, Power ISA Operating Environment Architecture. (The system floating-point enabled exception error handler is never invoked...
because of a disabled DFP exception.) The effects of the four possible settings of these bits are as follows.

<table>
<thead>
<tr>
<th>FE0</th>
<th>FE1</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Ignore Exceptions Mode</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Imprecise Nonrecoverable Mode</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Imprecise Recoverable Mode</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Precise Mode</td>
</tr>
</tbody>
</table>

In all cases, the question of whether a DFP result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler has been executed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. (Recall that, for the two Imprecise modes, the instruction at which the system floating-point enabled exception error handler is invoked need not be the instruction that caused the exception.) The instruction at which the system floating-point enabled exception error handler is invoked has not been executed unless it is the excepting instruction, in which case it has been executed if the exception is not among those listed on page 197 as suppressed.

**Programming Note**

In the ignore and both imprecise modes, a Floating-Point Status and Control Register instruction can be used to force any exceptions, due to instructions initiated before the Floating-Point Status and Control Register instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In either of the Imprecise modes, a Floating-Point Status and Control Register instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to instructions initiated before the Floating-Point Status and Control Register instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to zero.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to one for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to one.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

### 5.5.10.1 Invalid Operation Exception

**Definition**

An Invalid Operation Exception occurs when an operand is invalid for the specified DFP operation. The invalid DFP operations are:

- Any DFP operation on a signaling NaN (SNaN), except for Test, Round To DFP Short, Convert To DFP Long, Decode DPD To BCD, Extract Biased Exponent, Insert Biased Exponent, Shift Significand Left Immediate, and Shift Significand Right Immediate
For add or subtract operations, magnitude subtraction of infinities \((\infty + -\infty)\) and Invalid Operation Exception is enabled (\(VE=1\)) and Invalid Operation occurs, the following actions are taken:

1. **One or two Invalid Operation Exceptions are set:**
   - \(VXSNAN\) (if SNaN)
   - \(VXXS\) (if \(\infty \times \infty\))
   - \(VXDI\) (if \(\infty \div \infty\))
   - \(VXZDZ\) (if \(0 \div 0\))
   - \(VXIMZ\) (if \(\infty \times 0\))
   - \(VXVC\) (if invalid comp)
   - \(VXCVI\) (if invalid conversion)

2. **If the operation is an arithmetic, quantum-adjustment, conversion, or format:**
   - The target FPR is unchanged.
   - \(FR\) and \(FI\) are set to zero.
   - \(FPFR\) is set to indicate the class of the result (Quiet NaN)

3. **If the operation is a compare:**
   - \(FR\), \(FI\), and \(C\) are unchanged.
   - \(FPCC\) is set to reflect unordered.

When Invalid Operation Exception is disabled (\(VE=0\)) and Invalid Operation occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set:
   - \(VXSNAN\) (if SNaN)
   - \(VXXS\) (if \(\infty \times \infty\))
   - \(VXDI\) (if \(\infty \div \infty\))
   - \(VXZDZ\) (if \(0 \div 0\))
   - \(VXIMZ\) (if \(\infty \times 0\))
   - \(VXVC\) (if invalid comp)
   - \(VXCVI\) (if invalid conversion)

2. **If the operation is an arithmetic, quantum-adjustment, Round to DFP Long, Convert to DFP Extended, or format:**
   - The target FPR is set to a Quiet NaN.
   - \(FR\) and \(FI\) are set to zero.
   - \(FPFR\) is set to indicate the class of the result (Quiet NaN)

3. **If the operation is a Convert To Fixed:**
   - The target FPR is set as follows:
     - \(FRT\) is set to the most positive 64-bit binary integer if the operand in FRB is a positive or \(+\infty\), and to the most negative 64-bit binary integer if the operand in FRB is a negative number, \(-\infty\), or NaN.
     - \(FR\) and \(FI\) are set to zero.
     - \(FPFR\) is unchanged.

4. **If the operation is a compare:**
   - \(FR\), \(FI\), and \(C\) are unchanged.
   - \(FPCC\) is set to reflect unordered.

### 5.5.10.2 Zero Divide Exception

**Definition**

A Zero Divide Exception occurs when a Divide instruction is executed with a zero divisor value and a finite nonzero dividend value.

**Action**

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled (\(ZE=1\)) and Zero Divide occurs, the following actions are taken:

1. **Zero Divide Exception is set:**
   \(ZE \leftarrow 1\)

2. **The target FPR is unchanged:**

3. **\(FR\) and \(FI\) are set to zero:**

4. **\(FPFR\) is unchanged:**

When Zero Divide Exception is disabled (\(ZE=0\)) and Zero Divide occurs, the following actions are taken:

1. **Zero Divide Exception is set:**
   \(ZE \leftarrow 1\)

2. **The target FPR is set to \(\pm\infty\), where the sign is determined by the XOR of the signs of the operands:**

---

**Programming Note**

In addition, an Invalid Operation Exception occurs if software explicitly requests this by executing an \(mtfsi\), \(mtfsf\), or \(mtfsb1\) instruction that sets \(VXSOFT\) to 1 (Software Request). The purpose of \(VXSOFT\) is to allow software to cause an Invalid Operation Exception for a condition that is not necessarily associated with the execution of a DFP instruction. For example, it might be set by a program that computes a square root, if the source operand is negative.
3. \( FR \) and \( FL \) are set to zero
4. \( FPRF \) is set to indicate the class and sign of the result (±∞)

### 5.5.10.3 Overflow Exception

**Definition**

An overflow exception occurs whenever the target format’s largest finite number is exceeded in magnitude by what would have been the rounded result if the exponent range were unbounded.

**Action**

Except for Reround, the following describes the handling of the IEEE overflow exception condition. The Reround operation does not recognize an overflow exception condition.

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled (OE=1) and overflow occurs, the following actions are taken:

1. Overflow Exception is set \( OX \) ← 1
2. The infinitely precise result is divided by \( 10^{\alpha} \). That is, the exponent adjustment \( \alpha \) is subtracted from the exponent. This is called the *wrapped result*. The exponent adjustment for all operations, except for Round To DFP Short and Round To DFP Long, is 576 for DFP Long and 9216 for DFP Extended. For Round To DFP Short and Round To DFP Long, the exponent adjustment is 192 for the source format of DFP Long and 3072 for the source format of DFP Extended.
3. The wrapped result is rounded to the target-format precision. This is called the *wrapped rounded result*.
4. If the wrapped rounded result has only one form, it is the delivered result. If the wrapped rounded result has redundant forms and is exact, the result of the form that has the exponent closest to the wrapped ideal exponent is returned. If the wrapped rounded result has redundant forms and is inexact, the result of the form that has the smallest exponent is returned. The wrapped ideal exponent is the result of subtracting the exponent adjustment from the ideal exponent.
5. \( FPRF \) is set to indicate the class and sign of the result (± Normal Number)

When Overflow Exception is disabled (OE=0) and overflow occurs, the following actions are taken:

1. Overflow Exception is set \( OX \) ← 1
2. Inexact Exception is set \( XX \) ← 1
3. The result is determined by the rounding mode and the sign of the intermediate result as follows.

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>Sign of intermediate result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round to Nearest, Ties to Even</td>
<td>+∞</td>
</tr>
<tr>
<td>Round toward 0</td>
<td>( +N_{\text{long}} )</td>
</tr>
<tr>
<td>Round toward +∞</td>
<td>+∞</td>
</tr>
<tr>
<td>Round toward -∞</td>
<td>( -N_{\text{long}} )</td>
</tr>
<tr>
<td>Round to Nearest, Ties away from 0</td>
<td>+∞</td>
</tr>
<tr>
<td>Round to Nearest, Ties toward 0</td>
<td>+∞</td>
</tr>
<tr>
<td>Round away from 0</td>
<td>+∞</td>
</tr>
<tr>
<td>Round to prepare for shorter precision</td>
<td>( +N_{\text{long}} )</td>
</tr>
</tbody>
</table>

**Figure 75. Overflow Results When Exception Is Disabled**

4. The result is placed into the target FPR
5. \( FR \) is set to one if the returned result is ±∞, and is set to zero if the returned result is ±\( N_{\text{long}} \).
6. \( FL \) is set to one
7. \( FPRF \) is set to indicate the class and sign of the result (±∞ or ± Normal Number)

### 5.5.10.4 Underflow Exception

**Definition**

Except for Reround, the following describes the handling of the IEEE underflow exception condition. The Reround operation does not recognize an underflow exception condition.

The Underflow Exception is defined differently for the enabled and disabled states. However, a tininess condition is recognized in both states when a result computed as though both the precision and exponent range were unbounded would be nonzero and less than the target format’s smallest normal number, \( N_{\text{min}} \), in magnitude.

Unless otherwise defined in the instruction description, an underflow exception occurs as follows:

- **Enabled:**
  - When the tininess condition is recognized.
- **Disabled:**
  - When the tininess condition is recognized and when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.
**Action**

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled (UE=1) and underflow occurs, the following actions are taken:

1. Underflow Exception is set $UX \leftarrow 1$
2. The infinitely precise result is multiplied by $10^\alpha$. That is, the exponent adjustment $\alpha$ is added to the exponent. This is called the wrapped result. The exponent adjustment for all operations, except for Round To DFP Short and Round To DFP Long, is 576 for DFP Long and 9216 for DFP Extended. For Round To DFP Short and Round To DFP Long, the exponent adjustment is 192 for the source format of DFP Long and 3072 for the source format of DFP Extended.
3. The wrapped result is rounded to the target-format precision. This is called the wrapped rounded result.
4. If the wrapped rounded result has only one form, it is the delivered result. If the wrapped rounded result has redundant forms and is exact, the result of the form that has the exponent closest to the wrapped ideal exponent is returned. If the wrapped rounded result has redundant forms and is inexact, the result of the form that has the smallest exponent is returned. The wrapped ideal exponent is the result of adding the exponent adjustment to the ideal exponent.
5. $FPRF$ is set to indicate the class and sign of the result ($\pm$ Normal number)

When Underflow Exception is disabled (UE=0) and underflow occurs, the following actions are taken:

1. Underflow Exception is set $UX \leftarrow 1$
2. The infinitely precise result is rounded to the target-format precision.
3. The rounded result is returned. If this result has redundant forms, the result of the form that is closest to the ideal exponent is returned.
4. $FPRF$ is set to indicate the class and sign of the result ($\pm$ Normal number, $\pm$ Subnormal Number, or $\pm$ Zero)

**5.5.10.5 Inexact Exception**

**Definition**

Except for Round to FP Integer Without Inexact, the following describes the handling of the IEEE inexact exception condition. The Round to FP Integer Without Inexact does not recognize an inexact exception condition.

An Inexact Exception occurs when either of two conditions occur during rounding:

1. The delivered result differs from what would have been computed were both the precision and exponent range unbounded.
2. The rounded result overflows and Overflow Exception is disabled.

**Action**

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set $XX \leftarrow 1$
2. The rounded or overflowed result is placed into the target FPR
3. $FPRF$ is set to indicate the class and sign of the result

**Programming Note**

In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.
### 5.5.11 Summary of Normal Rounding And Range Actions

Figure 76 and Figure 77 summarize rounding and range actions, with the following exceptions:

- The *Reround* operation recognizes neither an underflow nor an overflow exception.
- The *Round to FP Integer Without Inexact* operation does not recognize the inexact operation exception.

#### Figure 76. Rounding and Range Actions (Part 1)

<table>
<thead>
<tr>
<th>Range of (v)</th>
<th>Case</th>
<th>Result (r) when Rounding Mode Is</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RNE</td>
</tr>
<tr>
<td>(v &lt; -Nmax), (q &lt; -Nmax)</td>
<td>Overflow</td>
<td>(-\infty)</td>
</tr>
<tr>
<td>(v &lt; -Nmax), (q = -Nmax)</td>
<td>Normal</td>
<td>(-Nmax)</td>
</tr>
<tr>
<td>(-Nmax \leq v \leq -Nmin)</td>
<td>Normal</td>
<td>(b)</td>
</tr>
<tr>
<td>(-Nmin &lt; v \leq -Dmin)</td>
<td>Tiny</td>
<td>(\times b^*)</td>
</tr>
<tr>
<td>(-Dmin &lt; v &lt; -Dmin/2)</td>
<td>Tiny</td>
<td>(-\times Dmin)</td>
</tr>
<tr>
<td>(v = -Dmin/2)</td>
<td>Tiny</td>
<td>(-\times 0)</td>
</tr>
<tr>
<td>(-Dmin/2 &lt; v &lt; 0)</td>
<td>Tiny</td>
<td>(-\times 0)</td>
</tr>
<tr>
<td>(v = 0)</td>
<td>EZD</td>
<td>(\times +0)</td>
</tr>
<tr>
<td>(0 &lt; v &lt; +Dmin/2)</td>
<td>Tiny</td>
<td>(\times +0)</td>
</tr>
<tr>
<td>(v = +Dmin/2)</td>
<td>Tiny</td>
<td>(\times +0)</td>
</tr>
<tr>
<td>(+Dmin/2 &lt; v &lt; +Dmin)</td>
<td>Tiny</td>
<td>(+\times Dmin)</td>
</tr>
<tr>
<td>(+Dmin \leq v &lt; +Nmin)</td>
<td>Tiny</td>
<td>(\times b^*)</td>
</tr>
<tr>
<td>(+Nmin \leq v \leq +Nmax)</td>
<td>Normal</td>
<td>(b)</td>
</tr>
<tr>
<td>(+Nmax &lt; v, q = +Nmax)</td>
<td>Normal</td>
<td>(+\times Nmax)</td>
</tr>
<tr>
<td>(+Nmax &lt; v, q &gt; +Nmax)</td>
<td>Overflow</td>
<td>(+\infty)</td>
</tr>
</tbody>
</table>

#### Explanation:

- This situation cannot occur.
- The normal result \(r\) is considered to have been incremented.
- The rounded value, in the extreme case, may be \(Nmin\). In this case, the exception conditions are underflow, inexact, and incremented.
- The value derived when the precise result \(v\) is rounded to the destination’s precision, including both bounded precision and bounded exponent range.
- The value derived when the precise result \(v\) is rounded to the destination’s precision, but assuming an unbounded exponent range.
- This is the returned value when neither overflow nor underflow is enabled.
- Precise result before rounding, assuming unbounded precision and an unbounded exponent range. For data-format conversion operations, \(v\) is the source value.
- Smallest (in magnitude) representable subnormal number in the target format.
- The result \(r\) of the exact-zero-difference case applies only to ADD and SUBTRACT with both source operands having opposite signs. (For ADD and SUBTRACT, when both source operands have the same sign, the sign of the zero result is the same sign as the sign of the source operands.)
- Largest (in magnitude) representable finite number in the target format.
- Smallest (in magnitude) representable normalized number in the target format.
- Round away from 0.
- Round to Prepare for Shorter Precision.
- Round to Nearest, Ties from away from 0.
- Round to Nearest, Ties to even.
- Round to Nearest, Ties toward 0.
- Round toward \(+\infty\).
- Round toward \(-\infty\).
- Round toward 0.
<table>
<thead>
<tr>
<th>Case</th>
<th>Is r inexact (rsv)</th>
<th>OE1</th>
<th>UE1</th>
<th>XE1</th>
<th>Is r incremented (r[p+q][v])</th>
<th>Is q inexact (qsv)</th>
<th>Is q incremented (q[p+q][v])</th>
<th>Returned Results and Status Setting*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overflow</td>
<td>Yes</td>
<td>No</td>
<td>—</td>
<td>No</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>T[x], OX=1, FI=0, FR=0, XX=1</td>
</tr>
<tr>
<td>Overflow</td>
<td>Yes</td>
<td>No</td>
<td>—</td>
<td>No</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>T[x], OX=1, FI=0, FR=1, XX=1</td>
</tr>
<tr>
<td>Overflow</td>
<td>Yes</td>
<td>No</td>
<td>—</td>
<td>Yes</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>T[x], OX=1, FI=0, FR=0, XX=1, TX</td>
</tr>
<tr>
<td>Overflow</td>
<td>Yes</td>
<td>No</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>T[x], OX=1, FI=0, FR=1, XX=1, TX</td>
</tr>
<tr>
<td>Overflow</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>No</td>
<td>—</td>
<td>No</td>
<td>T[x+b], OX=1, FI=0, FR=0, TO</td>
</tr>
<tr>
<td>Overflow</td>
<td>Yes</td>
<td>No</td>
<td>—</td>
<td>No</td>
<td>Yes</td>
<td>—</td>
<td>Yes</td>
<td>T[x+b], OX=1, FI=0, FR=1, XX=1, TO</td>
</tr>
<tr>
<td>Normal</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Normal</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>No</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>T[x], FI=0, FR=0</td>
</tr>
<tr>
<td>Normal</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>T[x], FI=1, FR=1, XX=1</td>
</tr>
<tr>
<td>Normal</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>Yes</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>T[x], FI=1, FR=0, XX=1, TX</td>
</tr>
<tr>
<td>Tiny</td>
<td>No</td>
<td>—</td>
<td>Yes</td>
<td>—</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>T[x+b], UX=1, FI=0, FR=0, TU</td>
</tr>
<tr>
<td>Tiny</td>
<td>Yes</td>
<td>—</td>
<td>No</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>T[x+b], UX=1, FI=0, FR=1, XX=1, TX</td>
</tr>
<tr>
<td>Tiny</td>
<td>Yes</td>
<td>—</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>T[x+b], UX=1, FI=1, FR=0, XX=1, TX</td>
</tr>
<tr>
<td>Tiny</td>
<td>Yes</td>
<td>—</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>T[x+b], UX=1, FI=1, FR=1, XX=1, TX</td>
</tr>
<tr>
<td>Tiny</td>
<td>Yes</td>
<td>—</td>
<td>Yes</td>
<td>—</td>
<td>No</td>
<td>—</td>
<td>No</td>
<td>T[x+b], UX=1, FI=0, FR=0, TO</td>
</tr>
<tr>
<td>Tiny</td>
<td>Yes</td>
<td>—</td>
<td>Yes</td>
<td>—</td>
<td>Yes</td>
<td>—</td>
<td>Yes</td>
<td>T[x+b], UX=1, FI=0, FR=1, XX=1, TU</td>
</tr>
<tr>
<td>Tiny</td>
<td>Yes</td>
<td>—</td>
<td>Yes</td>
<td>—</td>
<td>Yes</td>
<td>—</td>
<td>Yes</td>
<td>T[x+b], UX=1, FI=0, FR=1, XX=1, TU</td>
</tr>
</tbody>
</table>

Explanation:

— The results do not depend on this condition.
  1 This condition is true by virtue of the state of some condition to the left of this column.
  * Rounding sets only FI and FR. Setting of OX, XX, or UX is part of the exception actions. They are listed here for reference.
  β Wrap adjust, which depends on the type of operation and operand format. For all operations except Round to DFP Short and Round to DFP Long, the wrap adjust depends on the target format: β = 10^α, where α is 576 for DFP Long, and 9216 for DFP Extended. For Round to DFP Short and Round to DFP Long, the wrap adjust depends on the source format: β = 10^α where α is 192 for DFP Long and 3072 for DFP Extended.
  q The value derived when the precise result v is rounded to destination’s precision, but assuming an unbounded exponent range.
  r The result as defined in Part 1 of this figure.
  v Precise result before rounding, assuming unbounded precision and unbounded exponent range.
  FI Floating-Point-Fraction-Inexact status flag, FI. This status flag is non-sticky.
  FR Floating-Point-Fraction-Rounded status flag, FR.
  OX Floating-Point Overflow Exception status flag, OX.
  TO The system floating-point enabled exception error handler is invoked for the overflow exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.
  TU The system floating-point enabled exception error handler is invoked for the underflow exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.
  TX The system floating-point enabled exception error handler is invoked for the inexact exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.
  T[x] The value x is placed at the target operand location.
  TW(x) The wrapped rounded result x is placed at the target operand location. For all operations except data format conversions, the wrapped rounded result is in the same format and length as normal results at the target location. For data format conversions, the wrapped rounded result is in the same format and length as the source, but rounded to the target-format precision.
  UX Floating-Point-Underflow-Exception status bit.
  XX Floating-Point Inexact exception status bit. The flag is a sticky version of FI. When FI is set to a new value, the new value of XX is set to the result of ORing the old value of XX with the new value of FI.

Figure 77. Rounding and Range Actions (Part 2)
5.6 DFP Instruction Descriptions

The following sections describe the DFP instructions. When a 128-bit operand is used, it is held in a FPR pair and the instruction mnemonic uses a letter “q” to mean the quad-precision operation. Note that in the following descriptions, \( \text{FRX}_p \) denotes a FPR pair and must address an even-odd pair. If the \( \text{FRX}_p \) field specifies an odd-numbered register, then the instruction form is invalid. The notation \( \text{FRX}[p] \) means either a FPR, \( \text{FRX} \), or a FPR pair, \( \text{FRX}_p \).

For DFP instructions, if a DFP operand is returned, the trailing significand field of the target operand is encoded using preferred DPD codes.

5.6.1 DFP Arithmetic Instructions

All DFP arithmetic instructions are X-form instructions. They all set the FI and FR status flags, and also set the FPRF field. Furthermore, they all have an ideal exponent assigned and employ the record bit (Rc).

The arithmetic instructions consist of Add, Divide, Multiply, and Subtract.
**DFP Add X-form**

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
59 & FRT & FRA & FRB & 2 & Rc \\
\hline
0 & 6 & 11 & 16 & 21 & 31 \\
\hline
\end{array}
\]

**DFP Add Quad X-form**

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
63 & FRTp & FRAp & FRBp & 2 & Rc \\
\hline
0 & 6 & 11 & 16 & 21 & 31 \\
\hline
\end{array}
\]

The DFP operand in \( FRA[p] \) is added to the DFP operand in \( FRB[p] \).

The result is rounded to the target-format precision under control of \( DRN \) (bits 29:31 of the FPSCR). An appropriate form of the rounded result is selected based on the ideal exponent and is placed in \( FRT[p] \). The ideal exponent is the smaller exponent of the two source operands.

Figure 78 summarizes the actions for Add. Figure 78 does not include the setting of \( FPRF \). \( FPRF \) is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

\( dadd[q][] \) are treated as *Floating-Point* instructions in terms of resource availability.

**Special Registers Altered:**

- \( FPRF \)
- \( FR \)
- \( FI \)
- \( FX \)
- \( OX \)
- \( UX \)
- \( XX \)
- \( VXSNAN \)
- \( VXISI \)
- \( CR1 \) (if \( Rc=1 \))

**DFP Subtract X-form**

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
59 & FRT & FRA & FRB & 514 & Rc \\
\hline
0 & 6 & 11 & 16 & 21 & 31 \\
\hline
\end{array}
\]

**DFP Subtract Quad X-form**

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
63 & FRTp & FRAp & FRBp & 514 & Rc \\
\hline
0 & 6 & 11 & 16 & 21 & 31 \\
\hline
\end{array}
\]

The DFP operand in \( FRB[p] \) is subtracted from the DFP operand in \( FRA[p] \).

The result is rounded to the target-format precision under control of \( DRN \) (bits 29:31 of the FPSCR). An appropriate form of the rounded result is selected based on the ideal exponent and is placed in \( FRT[p] \). The ideal exponent is the smaller exponent of the two source operands.

The execution of Subtract is identical to that of Add, except that the operand in \( FRB \) participates in the operation with its sign bit inverted. See Figure 78. The table does not include the setting of \( FPRF \). \( FPRF \) is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

\( dsub[q][] \) are treated as *Floating-Point* instructions in terms of resource availability.

**Special Registers Altered:**

- \( FPRF \)
- \( FR \)
- \( FI \)
- \( FX \)
- \( OX \)
- \( UX \)
- \( XX \)
- \( VXSNAN \)
- \( VXISI \)
- \( CR1 \) (if \( Rc=1 \))
Operand a in FRA[p] is | Actions for Add \((a + b)\) when operand b in FRB[p] is
<table>
<thead>
<tr>
<th>(\infty)</th>
<th>(-\infty)</th>
<th>(+\infty)</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>T((-dINF))</td>
<td>T((-dINF))</td>
<td>VXISI: T((dNaN))</td>
<td>P(b)</td>
</tr>
<tr>
<td>(F)</td>
<td>T((-dINF))</td>
<td>S(a+b)</td>
<td>T((+dINF))</td>
<td>P(b)</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>VXISI: T((dNaN))</td>
<td>T((+dINF))</td>
<td>T((+dINF))</td>
<td>P(b)</td>
</tr>
<tr>
<td>QNaN</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
</tr>
<tr>
<td>SNaN</td>
<td>VXSNAN: U(a)</td>
<td>VXSNAN: U(a)</td>
<td>VXSNAN: U(a)</td>
<td>VXSNAN: U(a)</td>
</tr>
</tbody>
</table>

Explanation:
- \(a+b\) The value a added to b, rounded to the target-format precision and returned in the appropriate form. (See Section 5.5.11 on page 203)
- \(+dINF\) Default plus infinity.
- \(-dINF\) Default minus infinity.
- \(dNaN\) Default quiet NaN.
- \(F\) All finite numbers, including zeros.
- \(P(x)\) The QNaN of operand x is propagated and placed in FRT[p].
- \(S(x)\) The value x is placed in FRT[p] with the sign set by the rules of algebra. When the source operands have the same sign, the sign of the result is the same as the sign of the operands, including the case when the result is zero. When the operands have opposite signs, the sign of a zero result is positive in all rounding modes, except round toward \(-\infty\), in which case, the sign is minus.
- \(T(x)\) The value x is placed in FRT[p].
- \(U(x)\) The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
- VXISI: Floating-Point Invalid Operation (Infinity - Infinity) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)
- VXSNAN: Floating-Point Invalid Operation (SNaN) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)
DFP Multiply X-form

\[ \text{dmul} \text{ FRT,FRA,FRB } (R_c=0) \]
\[ \text{dmul.} \text{ FRT,FRA,FRB } (R_c=1) \]

DFP Multiply Quad X-form

\[ \text{dmulq} \text{ FRTp,FRAp,FRBp } (R_c=0) \]
\[ \text{dmulq.} \text{ FRTp,FRAp,FRBp } (R_c=1) \]

The DFP operand in \( \text{FRA}[p] \) is multiplied by the DFP operand in \( \text{FRB}[p] \).

The result is rounded to the target-format precision under control of \( \text{DRN} \) (bits 29:31 of the FPSCR). An appropriate form of the rounded result is selected based on the ideal exponent and is placed in \( \text{FRT}[p] \). The ideal exponent is the sum of the two exponents of the source operands.

The content is divided into sections about DFP Multiply X-form and DFP Multiply Quad X-form. Each section includes a table that lists the actions for Multiply, with columns for operands in \( \text{FRA}[p] \) and \( \text{FRB}[p] \), and actions for Multiply \( (a \times b) \) when operand \( b \) in \( \text{FRB}[p] \) is.

The table contains the following columns:
- **Operand a** in \( \text{FRA}[p] \)
- **Actions for Multiply** \( (a \times b) \) when operand \( b \) in \( \text{FRB}[p] \) is
- 0
- \( \infty \)
- \( \text{QNaN} \)
- \( \text{SNaN} \)

Each row in the table details the actions for different operations involving \( a \times b \) and \( b \) in \( \text{FRB}[p] \). The actions include:
- \( S(a \times b) \)
- \( S(a \times b) \text{ VXiNZ: T(dNaN)} \)
- \( S(a \times b) \text{ P(b)} \)
- \( VXSnan: U(b) \)
- \( P(a) \)
- \( P(a) \text{ VXiSNAN: U(a)} \)
- \( P(a) \text{ VXSnan: U(a)} \)
- \( P(a) \text{ VXSnan: U(a)} \)

The explanation for each action is provided, such as:
- \( S(a \times b) \) means the value is placed in \( \text{FRT}[p] \).
- \( P(a) \text{ VXiSNAN: U(a)} \) means the SNaN of operand \( a \) is converted to the corresponding QNaN and placed in \( \text{FRT}[p] \).

Figure 79 summarizes the actions for Multiply. Figure 79 does not include the setting of \( \text{FPRF} \). \( \text{FPRF} \) is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

\[ \text{dmulq}[q][] \] are treated as Floating-Point instructions in terms of resource availability.

### Special Registers Altered:

- \( \text{FPRF FR FI} \)
- \( \text{FX OX UX XX VXSnan VXImZ CR1} \) \( (\text{if } R_c=1) \)

Figure 79. Actions: Multiply
DFP Divide X-form

\[ \text{ddiv} \quad \text{FRT,FRA,FRB} \quad \text{(Rc=0)} \]
\[ \text{ddiv.} \quad \text{FRT,FRA,FRB} \quad \text{(Rc=1)} \]

DFP Divide Quad X-form

\[ \text{ddivq} \quad \text{FRTp,FRAp,FRBp} \quad \text{(Rc=0)} \]
\[ \text{ddivq.} \quad \text{FRTp,FRAp,FRBp} \quad \text{(Rc=1)} \]

The DFP operand in \( \text{FRA}[p] \) is divided by the DFP operand in \( \text{FRB}[p] \).

The result is rounded to the target-format precision under control of the DRN (bits 29:31 of the FPSCR). An appropriate form of the rounded result is selected based on the ideal exponent and is placed in \( \text{FRT}[p] \).

The ideal exponent is the difference of subtracting the exponent of the divisor from the exponent of the dividend.

Table: Actions for Divide (\( a \div b \)) when operand \( b \) in \( \text{FRB}[p] \) is

<table>
<thead>
<tr>
<th>Operand ( a ) in ( \text{FRA}[p] ) is</th>
<th>( 0 )</th>
<th>( \infty )</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>VXZDZ: T(dNaN)</td>
<td>S(a+b)</td>
<td>S(zt)</td>
<td>P(b)</td>
</tr>
<tr>
<td>( \infty )</td>
<td>Zx: S(dINF)</td>
<td>S(a+b)</td>
<td>S(zt)</td>
<td>P(b)</td>
</tr>
<tr>
<td>QNaN</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
</tr>
<tr>
<td>SNaN</td>
<td>VXSNAN: U(a)</td>
<td>VXSNAN: U(a)</td>
<td>VXSNAN: U(a)</td>
<td>VXSNAN: U(a)</td>
</tr>
</tbody>
</table>

Explanation:
- \( a \div b \) The value \( a \) divided by \( b \), rounded to the target-format precision and returned in the appropriate form. (See Section 5.5.11 on page 203.)
- \( dINF \) Default infinity.
- \( dNaN \) Default quiet NaN.
- \( Fx \) Finite nonzero number (includes both normal and subnormal numbers).
- \( P(x) \) The QNaN of operand \( x \) is propagated and placed in \( \text{FRT}[p] \).
- \( S(x) \) The value \( x \) is placed in \( \text{FRT}[p] \) with the sign set to the exclusive-OR of the source-operand signs.
- \( T(x) \) The value \( x \) is placed in \( \text{FRT}[p] \).
- \( U(x) \) The SNaN of operand \( x \) is converted to the corresponding QNaN and placed in \( \text{FRT}[p] \).
- \( \text{VXI DI:} \) Floating-Point Invalid Operation (Infinity \( \div \) Infinity) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199 for the exception actions.)
- \( \text{VXSNAN:} \) Floating-Point Invalid Operation (SNaN) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)
- \( \text{VXZDZ:} \) Floating-Point Invalid Operation (Zero \( \div \) Zero) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)
- \( \text{zt} \) True zero (zero significand and most negative exponent).
- \( Zx \) The Zero-Divide Exception occurs. The result is produced only when the exception is disabled (See Section 5.5.10.2 “Zero Divide Exception” on page 200.)
5.6.2 DFP Compare Instructions

The DFP compare instructions consist of the Compare Ordered and Compare Unordered instructions. The compare instructions do not provide the record bit. The comparison sets the designated CR field to indicate the result. FPCC is set in the same way.

The codes in the CR field BF and FPCC are defined for the DFP compare operations as follows.

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FL</td>
</tr>
<tr>
<td>1</td>
<td>FG</td>
</tr>
<tr>
<td>2</td>
<td>FE</td>
</tr>
<tr>
<td>3</td>
<td>FU</td>
</tr>
</tbody>
</table>

\( (\text{FRA}[p]) < (\text{FRB}[p]) \)
\( (\text{FRA}[p]) > (\text{FRB}[p]) \)
\( (\text{FRA}[p]) = (\text{FRB}[p]) \)
\( (\text{FRA}[p]) \neq (\text{FRB}[p]) \)
DFP Compare Unordered X-form

dcmpu BF,FRA,FRB

<table>
<thead>
<tr>
<th>59</th>
<th>BF</th>
<th>// FRA</th>
<th>FRB</th>
<th>642</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DFP Compare Unordered Quad X-form

dcmpuq BF,FRAp,FRBp

<table>
<thead>
<tr>
<th>63</th>
<th>BF</th>
<th>// FRAp</th>
<th>FRBp</th>
<th>642</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The DFP operand in FRA[p] is compared to the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the FPCC.

dcmpu[q] are treated as Floating-Point instructions in terms of resource availability.

Special Registers Altered:
- CR field BF
- FPCC
- FX VXSNaN

<table>
<thead>
<tr>
<th>Operand a in FRA[p] is</th>
<th>Actions for Compare Unordered (a:b) when operand b in FRB[p] is</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;o&gt;</td>
<td>F</td>
</tr>
<tr>
<td>o</td>
<td>AeqB</td>
</tr>
<tr>
<td>&lt;&gt; o</td>
<td>AeqB</td>
</tr>
<tr>
<td>o</td>
<td>AeqB</td>
</tr>
<tr>
<td>F</td>
<td>AeqB</td>
</tr>
<tr>
<td>&lt;o&gt;</td>
<td>AeqB</td>
</tr>
<tr>
<td>o</td>
<td>AeqB</td>
</tr>
<tr>
<td>F</td>
<td>AeqB</td>
</tr>
<tr>
<td>SNaN</td>
<td>Fu, VXSNaN</td>
</tr>
<tr>
<td>QNaN</td>
<td>Fu, VXSNaN</td>
</tr>
<tr>
<td>F</td>
<td>AeqB</td>
</tr>
<tr>
<td>&lt;o&gt;</td>
<td>AeqB</td>
</tr>
<tr>
<td>o</td>
<td>AeqB</td>
</tr>
<tr>
<td>F</td>
<td>AeqB</td>
</tr>
<tr>
<td>SNaN</td>
<td>Fu, VXSNaN</td>
</tr>
<tr>
<td>QNaN</td>
<td>Fu, VXSNaN</td>
</tr>
<tr>
<td>F</td>
<td>AeqB</td>
</tr>
<tr>
<td>&lt;o&gt;</td>
<td>AeqB</td>
</tr>
<tr>
<td>o</td>
<td>AeqB</td>
</tr>
<tr>
<td>F</td>
<td>AeqB</td>
</tr>
<tr>
<td>SNaN</td>
<td>Fu, VXSNaN</td>
</tr>
<tr>
<td>QNaN</td>
<td>Fu, VXSNaN</td>
</tr>
<tr>
<td>F</td>
<td>AeqB</td>
</tr>
<tr>
<td>&lt;o&gt;</td>
<td>AeqB</td>
</tr>
<tr>
<td>o</td>
<td>AeqB</td>
</tr>
<tr>
<td>F</td>
<td>AeqB</td>
</tr>
<tr>
<td>SNaN</td>
<td>Fu, VXSNaN</td>
</tr>
<tr>
<td>QNaN</td>
<td>Fu, VXSNaN</td>
</tr>
<tr>
<td>F</td>
<td>AeqB</td>
</tr>
<tr>
<td>&lt;o&gt;</td>
<td>AeqB</td>
</tr>
<tr>
<td>o</td>
<td>AeqB</td>
</tr>
<tr>
<td>F</td>
<td>AeqB</td>
</tr>
<tr>
<td>SNaN</td>
<td>Fu, VXSNaN</td>
</tr>
<tr>
<td>QNaN</td>
<td>Fu, VXSNaN</td>
</tr>
</tbody>
</table>

Explanation:
- C(a:b) Algebraic comparison. See the table below.
- F All finite numbers, including zeros.
- AeqB CR field BF and FPCC are set to 0b0010.
- AgtB CR field BF and FPCC are set to 0b0100.
- AltB CR field BF and FPCC are set to 0b1000.
- AuoB CR field BF and FPCC are set to 0b0001.
- VXSNaN Floating-Point Invalid Operation (SNaN) exception occurs. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)

<table>
<thead>
<tr>
<th>Relation of Value a to Value b</th>
<th>Action for C(a:b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b</td>
<td>AeqB</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>AltB</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>AgtB</td>
</tr>
</tbody>
</table>

Figure 81. Actions: Compare Unordered
**DFP Compare Ordered X-form**

dcmpo BF,FRA,FRB

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>BF</td>
<td>//</td>
<td>FRA</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>130</td>
<td>21</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

**DFP Compare Ordered Quad X-form**

dcmpoq BF,FRAp,FRBp

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>BF</td>
<td>//</td>
<td>FRAp</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>130</td>
<td>21</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

The DFP operand in FRA[p] is compared to the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the FPCC.

dcmpo[q] are treated as Floating-Point instructions in terms of resource availability.

**Special Registers Altered:**
- CR field BF
- FPCC
- FX VXSNAN VXVC

---

**Operand a in FRA[p] is**

**Actions for Compare ordered a:b when operand b in FRB[p] is**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;-∞</td>
<td>F</td>
<td>+∞</td>
<td>QNaN</td>
</tr>
<tr>
<td>&lt;-∞</td>
<td>AeqB</td>
<td>AtB</td>
<td>AeqB</td>
<td>AuoB, VXVC</td>
</tr>
<tr>
<td>F</td>
<td>AgtB</td>
<td>C(a:b)</td>
<td>AtB</td>
<td>AuoB, VXVC</td>
</tr>
<tr>
<td>+∞</td>
<td>AgtB</td>
<td>AeqB</td>
<td>AeqB</td>
<td>AuoB, VXVC</td>
</tr>
<tr>
<td>QNaN</td>
<td>AuoB, VXVC</td>
<td>AuoB, VXVC</td>
<td>AuoB, VXVC</td>
<td>AuoB, VXSV</td>
</tr>
<tr>
<td>SNaN</td>
<td>AuoB, VXSV</td>
<td>AuoB, VXSV</td>
<td>AuoB, VXSV</td>
<td>AuoB, VXSV</td>
</tr>
</tbody>
</table>

**Explanation:**
- **C(a:b)**: Algebraic comparison. See the table below.
- **F**: All finite numbers, including zeros
- **AeqB**: CR field BF and FPCC are set to 0b0010.
- **AgtB**: CR field BF and FPCC are set to 0b0100.
- **AtB**: CR field BF and FPCC are set to 0b1000.
- **AuoB**: CR field BF and FPCC are set to 0b0001.
- **VXSV**: Floating-Point Invalid Operation (SNaN) exception occurs. Additionally, if the exception is disabled (VE=0), then VXVC is also set to one. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)
- **VXVC**: Floating-Point Invalid Operation (Invalid Compare) exception occurs. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)

---

<table>
<thead>
<tr>
<th>Relation of Value a to Value b</th>
<th>Action for C(a:b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b</td>
<td>AeqB</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>AtB</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>AgtB</td>
</tr>
</tbody>
</table>

**Figure 82. Actions: Compare Ordered**
5.6.3 DFP Test Instructions

The DFP test instructions consist of the Test Data Class, Test Data Group, Test Exponent, and Test Significance instructions, and they do not provide the record bit. The test instructions set the designated CR field to indicate the result. The FPSCR\textsubscript{FPCC} is set in the same way.

**DFP Test Data Class Z22-form**

\[
\begin{array}{c|c|c|c|c|c|c}
59 & BF & FRA & DCM & 194 & \\
0 & 6 & 9 & 11 & 16 & 21 & \\
\end{array}
\]

**DFP Test Data Class Quad Z22-form**

\[
\begin{array}{c|c|c|c|c|c|c}
63 & BF & FRAp & DCM & 194 & \\
0 & 6 & 9 & 11 & 16 & 21 & \\
\end{array}
\]

Let the DCM (Data Class Mask) field specify one or more of the 6 possible data classes, where each bit corresponds to a specific data class.

<table>
<thead>
<tr>
<th>DCM Bit</th>
<th>Data Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>1</td>
<td>Subnormal</td>
</tr>
<tr>
<td>2</td>
<td>Normal</td>
</tr>
<tr>
<td>3</td>
<td>Infinity</td>
</tr>
<tr>
<td>4</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>5</td>
<td>Signaling NaN</td>
</tr>
</tbody>
</table>

CR field BF and FPCC are set to indicate the sign of the DFP operand in FRA\textsubscript{p} and whether the data class of the DFP operand in FRA\textsubscript{p} matches any of the data classes specified by DCM.

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Operand positive with no match</td>
</tr>
<tr>
<td>0010</td>
<td>Operand positive with match</td>
</tr>
<tr>
<td>1000</td>
<td>Operand negative with no match</td>
</tr>
<tr>
<td>1010</td>
<td>Operand negative with match</td>
</tr>
</tbody>
</table>

**DFP Test Data Group Z22-form**

\[
\begin{array}{c|c|c|c|c|c|c}
59 & BF & FRA & DGM & 226 & \\
0 & 6 & 9 & 11 & 16 & 21 & \\
\end{array}
\]

**DFP Test Data Group Quad Z22-form**

\[
\begin{array}{c|c|c|c|c|c|c}
63 & BF & FRAp & DGM & 226 & \\
0 & 6 & 9 & 11 & 16 & 21 & \\
\end{array}
\]

Let the DGM (Data Group Mask) field specify one or more of the 6 possible data groups, where each bit corresponds to a specific data group.

The term extreme exponent means either the maximum exponent, \(x_{\text{max}}\), or the minimum exponent, \(x_{\text{min}}\).

<table>
<thead>
<tr>
<th>DGM Bit</th>
<th>Data Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero with non-extreme exponent</td>
</tr>
<tr>
<td>1</td>
<td>Zero with extreme exponent</td>
</tr>
<tr>
<td>2</td>
<td>Subnormal or (Normal with extreme exponent)</td>
</tr>
<tr>
<td>3</td>
<td>Normal with non-extreme exponent and leftmost zero digit in significand</td>
</tr>
<tr>
<td>4</td>
<td>Normal with non-extreme exponent and leftmost nonzero digit in significand</td>
</tr>
<tr>
<td>5</td>
<td>Special symbol (Infinity, QNaN, or SNaN)</td>
</tr>
</tbody>
</table>

CR field BF and FPCC are set to indicate the sign of the DFP operand in FRA\textsubscript{p} and whether the data group of the DFP operand in FRA\textsubscript{p} matches any of the data groups specified by DGM.

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Operand positive with no match</td>
</tr>
<tr>
<td>0010</td>
<td>Operand positive with match</td>
</tr>
<tr>
<td>1000</td>
<td>Operand negative with no match</td>
</tr>
<tr>
<td>1010</td>
<td>Operand negative with match</td>
</tr>
</tbody>
</table>

**dfstdc\textsubscript{q}** are treated as Floating-Point instructions in terms of resource availability.

**Special Registers Altered:**

- CR field BF
- FPCC

**dfstdg\textsubscript{q}** are treated as Floating-Point instructions in terms of resource availability.

**Special Registers Altered:**

- CR field BF
- FPCC
**DFP Test Exponent X-form**

dftstex BF,FRA,FRB

<table>
<thead>
<tr>
<th></th>
<th>BF</th>
<th>FRA</th>
<th>FRB</th>
<th>162</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**DFP Test Exponent Quad X-form**

dftstexq BF,FRAp,FRBp

<table>
<thead>
<tr>
<th></th>
<th>BF</th>
<th>FRA</th>
<th>FRB</th>
<th>162</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

The exponent value (Ea) of the DFP operand in FRA[p] is compared to the exponent value (Eb) of the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the FPCC.

The codes in the CR field BF and FPCC are defined for the DFP Test Exponent operations as follows.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ea &lt; Eb</td>
</tr>
<tr>
<td>1</td>
<td>Ea &gt; Eb</td>
</tr>
<tr>
<td>2</td>
<td>Ea = Eb</td>
</tr>
<tr>
<td>3</td>
<td>Ea ? Eb</td>
</tr>
</tbody>
</table>

Special Registers Altered:
- CR field BF
- FPCC

<table>
<thead>
<tr>
<th>Operand a in FRA[p] is</th>
<th>Actions for Test Exponent (Ea:Eb) when operand b in FRB[p] is</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>C(Ea:Eb)</td>
</tr>
<tr>
<td>∞</td>
<td>AuoB</td>
</tr>
<tr>
<td>QNaN</td>
<td>AuoB</td>
</tr>
<tr>
<td>SNaN</td>
<td>AuoB</td>
</tr>
</tbody>
</table>

Explanation:
- \( C(Ea:Eb) \) Algebraic comparison. See the table below.
- \( F \) All finite numbers, including zeros
- \( AeqB \) CR field BF and FPCC are set to 0b0010.
- \( AgtB \) CR field BF and FPCC are set to 0b0100.
- \( AltB \) CR field BF and FPCC are set to 0b1000.
- \( AuoB \) CR field BF and FPCC are set to 0b0001.

Relation of Value Ea to Value Eb

<table>
<thead>
<tr>
<th>Relation of Value Ea to Value Eb</th>
<th>Action for ( C(Ea:Eb) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Ea = Eb )</td>
<td>AeqB</td>
</tr>
<tr>
<td>( Ea &lt; Eb )</td>
<td>AltB</td>
</tr>
<tr>
<td>( Ea &gt; Eb )</td>
<td>AgtB</td>
</tr>
</tbody>
</table>

Figure 83. Actions: Test Exponent
**DFP Test Significance X-form**

\[ \text{dtsstf} \text{ BF,FRA,FRB} \]

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( k \neq 0 ) and ( k &lt; \text{NSDb} )</td>
</tr>
<tr>
<td>1</td>
<td>( k \neq 0 ) and ( k &gt; \text{NSDb} ), or ( k = 0 )</td>
</tr>
<tr>
<td>2</td>
<td>( k = 0 ) and ( k = \text{NSDb} )</td>
</tr>
<tr>
<td>3</td>
<td>( k &lt; \text{NSDb} )</td>
</tr>
</tbody>
</table>

**DFP Test Significance Quad X-form**

\[ \text{dtsstfq} \text{ BF,FRA,FRBp} \]

Let \( k \) be the contents of bits 58:63 of \( \text{FPR}[\text{FRA}] \) that specifies the reference significance.

For \( \text{dtsstf} \), let the value \( \text{NSDb} \) be the number of significant digits of the DFP value in \( \text{FPR}[\text{FRB}] \).

For \( \text{dtsstfq} \), let the value \( \text{NSDb} \) be the number of significant digits of the DFP value in \( \text{FPR}[\text{FRBp:FRBp+1}] \).

For this instruction, the number of significant digits of the value 0 is considered to be zero.

\( \text{NSDb} \) is compared to \( k \). The result of the compare is placed into CR field \( \text{BF} \) and the \( \text{FPCC} \) as follows.

- \( k \neq 0 \) and \( k < \text{NSDb} \) : \( \text{AeqB} \)
- \( k \neq 0 \) and \( k > \text{NSDb} \), or \( k = 0 \) : \( \text{AgtB} \)
- \( k = 0 \) and \( k = \text{NSDb} \) : \( \text{AltB} \)

**Actions for Test Significance**

<table>
<thead>
<tr>
<th>Relation of Value ( \text{NSDb} ) to Value ( k )</th>
<th>Action for ( \text{C}(k:\text{NSDb}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k \neq 0 ) and ( k &lt; \text{NSDb} )</td>
<td>( \text{AeqB} )</td>
</tr>
<tr>
<td>( k \neq 0 ) and ( k &gt; \text{NSDb} ), or ( k = 0 )</td>
<td>( \text{AgtB} )</td>
</tr>
<tr>
<td>( k = 0 ) and ( k = \text{NSDb} )</td>
<td>( \text{AltB} )</td>
</tr>
</tbody>
</table>

**Programming Note**

The reference significance can be loaded into a FPR using a Load Float as Integer Word Algebraic instruction.
DFP Test Significance Immediate X-form

dtstsfi BF,UIM,FRB

<table>
<thead>
<tr>
<th>59</th>
<th>BF</th>
<th>UIM</th>
<th>FRB</th>
<th>675</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>10</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

DFP Test Significance Immediate Quad X-form

dtstsfq BF,UIM,FRBp

<table>
<thead>
<tr>
<th>63</th>
<th>BF</th>
<th>UIM</th>
<th>FRBp</th>
<th>675</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>10</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

Let the value UIM specify the reference significance.

For dtstsfi, let the value NSDb be the number of significant digits of the DFP value in FPR[FRB].

For dtstsfq, let the value NSDb be the number of significant digits of the DFP value in FPR[FRBp:FRBp+1].

For this instruction, the number of significant digits of the value 0 is considered to be zero.

NSDb is compared to UIM. The result of the compare is placed into CR field BF and the FPCC as follows.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UIM ≠ 0 and UIM &lt; NSDb</td>
</tr>
<tr>
<td>1</td>
<td>UIM ≠ 0 and UIM &gt; NSDb, or UIM = 0</td>
</tr>
<tr>
<td>2</td>
<td>UIM ≠ 0 and UIM = NSDb</td>
</tr>
<tr>
<td>3</td>
<td>UIM ? NSDb</td>
</tr>
</tbody>
</table>

dtstsfi(q) are treated as Floating-Point instructions in terms of resource availability.

Special Registers Altered:
- CR field BF
- FPCC

Actions for Test Significance when the operand in VSR[FRB] or VSR[FRBp:FRBp+1] is

<table>
<thead>
<tr>
<th>F</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(UIM:NSDb)</td>
<td>AeqB</td>
<td>AeqB</td>
</tr>
</tbody>
</table>

Explanation:

C(UIM:NSDb) Algebraic comparison. See the table below.

- F
- AeqB CR field BF and FPCC are set to 0b0010.
- AgtB CR field BF and FPCC are set to 0b0100.
- AltB CR field BF and FPCC are set to 0b1000.
- AuoB CR field BF and FPCC are set to 0b0001.

<table>
<thead>
<tr>
<th>Relation of Value NSDb to Value UIM</th>
<th>Action for C(UIM:NSDb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIM ≠ 0 and UIM = NSDb</td>
<td>AeqB</td>
</tr>
<tr>
<td>UIM ≠ 0 and UIM &lt; NSDb</td>
<td>AltB</td>
</tr>
<tr>
<td>UIM ≠ 0 and UIM &gt; NSDb, or UIM = 0</td>
<td>AgtB</td>
</tr>
</tbody>
</table>

Figure 85. Actions: Test Significance
5.6.4 DFP Quantum Adjustment Instructions

The Quantum Adjustment operations consist of the Quantize, Quantize Immediate, Reround, and Round To FP Integer operations.

The Quantum Adjustment instructions are Z23-form instructions and have an immediate RMC (Rounding-Mode-Control) field, which specifies the rounding mode used. For Quantize, Quantize Immediate, and Reround, the RMC field contains the primary encoding. For Round to FP Integer, the field contains either primary or secondary encoding, depending on the setting of a RMC-encoding-selection bit. See Section 5.5.2 “Rounding Mode Specification” on page 195 for the definition of RMC encoding.

DFP Quantize Immediate Z23-form

dquai TE,FRT,FRB,RMC
\( (Rc=0) \)
dquai. TE,FRT,FRB,RMC
\( (Rc=1) \)

DFP Quantize Immediate Quad Z23-form

dquaiz TE,FRTp,FRBp,RMC
\( (Rc=0) \)
dquaiz. TE,FRTp,FRBp,RMC
\( (Rc=1) \)

The DFP operand in \( FRB[p] \) is converted and rounded to the form with the exponent specified by \( TE \) based on the rounding mode specified in the RMC field. \( TE \) is a 5-bit signed binary integer. The result of that form is placed in \( FRT[p] \). The sign of the result is the same as the sign of the operand in \( FRB[p] \). The ideal exponent is the exponent specified by \( TE \).

When the value of the operand in \( FRB[p] \) is greater than \((10^p-1) \times 10^{TE} \), where \( p \) is the format precision, an invalid operation exception is recognized.

When the delivered result differs in value from the operand in \( FRB[p] \), an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in \( FRB[p] \).

\( FPRF \) is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

\( dquaiz[q][] \) are treated as Floating-Point instructions in terms of resource availability.

Special Registers Altered:

\[ \begin{align*}
FPRF & \quad FR \quad FI \\
FX & \quad XX \\
VXSNAN & \quad VXCVI \\
CR1 & \quad (if Rc=1)
\end{align*} \]
DFP Quantize Immediate can be used to adjust values to a form having the specified exponent in the range -16 to 15. If the adjustment requires the significand to be shifted left, then:

- if the result would cause overflow from the most significant digit, the result is a default QNaN;
- otherwise the result is the adjusted value (left shifted with matching exponent).

If the adjustment requires the significand to be shifted right, the result is rounded based on the value of the RMC field.

DFP Quantize Immediate can round a value to a specific number of fractional digits. Consider the computation of sales tax. Values expressed in U.S. dollars have 2 fractional digits, and sales tax rates typically have 3 fractional digits. The product of value and rate will yield 5 fractional digits. For example:

\[ 39.95 \times 0.075 = 2.99625 \]

This result needs to be rounded to the penny to compute the correct tax of $3.00.

The following sequence computes the sales tax assuming the pre-tax total is in FRA and the tax rate is in FRB. The DFP Quantize Immediate instruction rounds the product (FRA \times FRB) to 2 fractional digits (TE = -2) using Round to nearest, ties away from 0 (RMC = 2). The quantized and rounded result is placed in FRT.

```
dmul f0,FRA,FRB
dqua -2,FRT,f0,2
```
**DFP Quantize Z23-form**

\[ \text{dqua.} \quad \text{FRT, FRA, FRB, RMC} \quad (\text{Rc}=0) \]
\[ \text{dqua} \quad \text{FRT, FRA, FRB, RMC} \quad (\text{Rc}=1) \]

The DFP operand in register \( \text{FRB}[p] \) is converted and rounded to the form with the same exponent as that of the DFP operand in \( \text{FRA}[p] \) based on the rounding mode specified by \( \text{RMC} \). The sign of the result is the same as the sign of the operand in \( \text{FRB}[p] \). The ideal exponent is the exponent specified in \( \text{FRA}[p] \).

When the value of the operand in \( \text{FRB}[p] \) is greater than \( (10^p - 1) \times 10^{E_a} \), where \( p \) is the format precision and \( E_a \) is the exponent of the operand in \( \text{FRA}[p] \), an invalid operation exception is recognized.

When the delivered result differs in value from the operand in \( \text{FRB}[p] \), an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in \( \text{FRB}[p] \).

**Programming Note**

**DFP Quantize** can be used to adjust one DFP value (\( \text{FRB}[p] \)) to a form having the same exponent as a second DFP value (\( \text{FRA}[p] \)). If the adjustment requires the significand to be shifted left, then:
- if the result would cause overflow from the most significant digit, the result is a default QNaN;
- otherwise the result is the adjusted value (left shifted with matching exponent).

If the adjustment requires the significand to be shifted right, the result is rounded based on the value of \( \text{RMC} \). Figure 86 shows examples of these adjustments.

---

**DFP Quantize Quad Z23-form**

\[ \text{dqua}\text{q} \quad \text{FRTp, FRAp, FRBp, RMC} \quad (\text{Rc}=0) \]
\[ \text{dqua}\text{q} \quad \text{FRTp, FRAp, FRBp, RMC} \quad (\text{Rc}=1) \]

Figure 87 and Figure 88 summarize the actions. The tables do not include the setting of \( \text{FPRF} \). \( \text{FPRF} \) is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

\[ \text{dqua}[[q]] \] are treated as Floating-Point instructions in terms of resource availability.

**Special Register Altered:**

- \( \text{FPRF} \)
- \( \text{FRA} \)
- \( \text{FRB} \)
- \( \text{FRT} \)
- \( \text{FX} \)
- \( \text{XX} \)
- \( \text{VXSNAN} \)
- \( \text{VXCVI} \)
- \( \text{CR1} \) (if \( \text{Rc}=1 \))

---

<table>
<thead>
<tr>
<th>FRA</th>
<th>FRB</th>
<th>FRT when RMC=1</th>
<th>FRT when RMC=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( (1 \times 10^0) )</td>
<td>9 ( (9 \times 10^4) )</td>
<td>9 ( (9 \times 10^5) )</td>
<td>9 ( (9 \times 10^5) )</td>
</tr>
<tr>
<td>1.00 ( (100 \times 10^{-2}) )</td>
<td>9 ( (9 \times 10^5) )</td>
<td>9.00 ( (900 \times 10^{-2}) )</td>
<td>9.00 ( (900 \times 10^{-2}) )</td>
</tr>
<tr>
<td>1 ( (1 \times 10^0) )</td>
<td>49.1234 ( (491234 \times 10^{-4}) )</td>
<td>49 ( (49 \times 10^5) )</td>
<td>49 ( (49 \times 10^5) )</td>
</tr>
<tr>
<td>1.00 ( (100 \times 10^{-2}) )</td>
<td>49.1234 ( (491234 \times 10^{-4}) )</td>
<td>49.12 ( (4912 \times 10^{-2}) )</td>
<td>49.12 ( (4912 \times 10^{-2}) )</td>
</tr>
<tr>
<td>1 ( (1 \times 10^0) )</td>
<td>49.9876 ( (499876 \times 10^{-4}) )</td>
<td>50 ( (50 \times 10^0) )</td>
<td>50 ( (50 \times 10^0) )</td>
</tr>
<tr>
<td>1.00 ( (100 \times 10^{-2}) )</td>
<td>49.9876 ( (499876 \times 10^{-4}) )</td>
<td>49.98 ( (4998 \times 10^{-2}) )</td>
<td>49.99 ( (4999 \times 10^{-2}) )</td>
</tr>
<tr>
<td>0.01 ( (1 \times 10^{-2}) )</td>
<td>49.9876 ( (499876 \times 10^{-4}) )</td>
<td>49.98 ( (4998 \times 10^{-2}) )</td>
<td>49.99 ( (4999 \times 10^{-2}) )</td>
</tr>
<tr>
<td>1 ( (1 \times 10^0) )</td>
<td>9999999999999999 ( (9999999999999999 \times 10^0) )</td>
<td>9999999999999999 ( (9999999999999999 \times 10^0) )</td>
<td>9999999999999999 ( (9999999999999999 \times 10^0) )</td>
</tr>
<tr>
<td>1.0 ( (10 \times 10^{-2}) )</td>
<td>9999999999999999 ( (9999999999999999 \times 10^0) )</td>
<td>QNaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

Figure 86. DFP Quantize examples
### Actions for Quantize when operand b in FRB[p] is

<table>
<thead>
<tr>
<th>Operand a in FRA[p] is</th>
<th>Actions for Quantize when operand b in FRB[p] is</th>
<th>0</th>
<th>Fn</th>
<th>∞</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td>VXCVI: T(dNaN)</td>
<td>P(b)</td>
</tr>
<tr>
<td>Fn</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td>VXCVI: T(dNaN)</td>
<td>P(b)</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td>VXCVI: T(dNaN)</td>
<td>P(b)</td>
</tr>
<tr>
<td>QNaN</td>
<td></td>
<td>P(a)</td>
<td></td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
</tr>
<tr>
<td>SNaN</td>
<td></td>
<td>VXSNAN: U(a)</td>
<td></td>
<td>VXSNAN: U(a)</td>
<td>VXSNAN: U(a)</td>
<td>VXSNAN: U(a)</td>
</tr>
</tbody>
</table>

**Explanation:**

* See next table.

- **dINF** Default infinity
- **dNaN** Default quiet NaN
- **Fn** Finite nonzero numbers (includes both subnormal and normal numbers)
- **P(x)** The QNaN of operand x is propagated and placed in FRT[p]
- **T(x)** The value x is placed in FRT[p]
- **U(x)** The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
- **VXCVI**: Floating-Point Invalid Operation (Invalid Conversion) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)
- **VXSNAN**: Floating-Point Invalid Operation (SNaN) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)

### Actions for Quantize when operand b in FRB[p] is

<table>
<thead>
<tr>
<th>Te &lt; Se</th>
<th>Actions for Quantize when operand b in FRB[p] is</th>
<th>0</th>
<th>Fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0 &gt; (10^p - 1) \times 10^{Te}</td>
<td>E(0)</td>
<td>VXCVI: T(dNaN)</td>
<td></td>
</tr>
<tr>
<td>V0 ≤ (10^p - 1) \times 10^{Te}</td>
<td>E(0)</td>
<td>L(b)</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

- **dNaN** Default quiet NaN
- **E(0)** The value of zero with the exponent value Te is placed in FRT[p].
- **L(x)** The operand x is converted to the form with the exponent value Te.
- **p** The precision of the format.
- **QR(x)** The operand x is rounded to the result of the form with the exponent value Te based on the specified rounding mode. The result of that form is placed in FRT[p].
- **Se** The exponent of the operand in FRB[p].
- **Te** The target exponent; FRA[p] for dqua[q], or TE, a 5-bit signed binary integer for dqua[q].
- **T(x)** The value x is placed in FRT[p].
- **Vb** The value of the operand in FRB[p].
- **W(x)** The value and the form of operand x is placed in FRT[p].
- **VXCVI**: Floating-Point Invalid Operation (Invalid Conversion) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)
**DFP Reround Z23-form**

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>FRA</th>
<th>FRB</th>
<th>RMC</th>
<th>35</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>23</td>
<td>31</td>
</tr>
</tbody>
</table>

**DFP Reround Quad Z23-form**

<table>
<thead>
<tr>
<th>63</th>
<th>FRTp</th>
<th>FRA</th>
<th>FRBp</th>
<th>RMC</th>
<th>35</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>23</td>
<td>31</td>
</tr>
</tbody>
</table>

Let $k$ be the contents of bits 58:63 of FRA that specifies the reference significance.

When the DFP operand in FRB[p] is a finite number, and if the reference significance is zero, or if the reference significance is nonzero and the number of significant digits of the source operand is less than or equal to the reference significance, then the value and the form of the source operand is placed in FRT[p]. If the reference significance is nonzero and the number of significant digits of the source operand is greater than the reference significance, then the source operand is converted and rounded to the number of significant digits specified in the reference significance based on the rounding mode specified in the RMC field. The result of the form with the specified number of significant digits is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p].

For this instruction, the number of significant digits of the value 0 is considered to be zero. The ideal exponent is the greater value of the exponent of the operand in FRB[p] and the referenced exponent. The referenced exponent is the resultant exponent if the operand in FRB[p] would have been converted and rounded to the number of significant digits specified in the reference significance based on the rounding mode specified by RMC.

If the exponent of the rounded result of the form that has the specified number of significant digits would be greater than $X_{max}$, an invalid operation exception (VXCVI) occurs. When the invalid-operation exception occurs, and if the exception is disabled, a default QNaN is returned. When an invalid-operation exception occurs, no inexact exception is recognized.

In the absence of an invalid-operation exception, if the result differs in value from the operand in FRB[p], an inexact exception is recognized.

This operation causes neither an overflow nor an underflow exception.

Figure 90 summarizes the actions for Reround. The table does not include the setting of FPRF. FPRF is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

drrnd[q][.] are treated as Floating-Point instructions in terms of resource availability.

**Special Registers Altered:**

- FPRF FR FI
- FX XX
- VXSNAN VXCVI
- CR1 (if $Rc=1$)

**Programming Note**

**DFP Reround** can be used to adjust a DFP value (FRB[p]) to have no more than a specified number (FRA[p]58:63) of significant digits. The result (FRT[p]) is right-justified leaving the specified number of digits and rounded as specified by RMC. If rounding increases the number of significant digits, the result is adjusted again (the significand is shifted right 1 digit and the exponent is incremented by 1). Figure 89 has example results from **DFP Reround** for 1, 2, and 10 significant digits.
Programming Note

DFP Reround is primarily used to round a DFP value to a specific number of digits before conversion to string format for printing or display. Another use for DFP Reround is to obtain the effective exponent of the most significant digit by specifying a reference significance of 1. The exponent can be extracted and used to compute the number of significant digits or to left-justify a value.

For example, the following sequence computes the number of significant digits and returns it as an integer. FRB is the DFP value for which we want the number of significant digits; f13 contains the reference significance value 0x0000000000000001; and r1 is the stack pointer, with free space for doublewords at offsets -8 and -16. These doublewords are used to transfer the biased exponents from the FPRs to GPRs for integer computation. R3 contains the result of $E(reround(1, \text{FRA}) - E(\text{FRA}) + 1$, where $E(x)$ represents the biased exponent of $x$.

```assembly
dxex f0,FRB
stfd f0,-16(r1)
drrnd f1,f13,FRB,1 # reround 1 digit toward 0
dxex f1,f1
stfd f1,-8(r1)
lfd r11,-16(r1)
lfd r3,-8(r1)
subf r3,r11,r3
addi r3,r3,1
```

Given the value 412.34 the result is $E(4 \times 10^2) - E(41234 \times 10^{-2}) + 1 = (398+2) - (398-2) + 1 = 400 - 396 + 1 = 5$. Additional code is required to detect and handle special values like Subnormal, Infinity, and NAN.

<table>
<thead>
<tr>
<th>FRA58:63 (binary)</th>
<th>FRB</th>
<th>FRT when RMC=1</th>
<th>FRT when RMC=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.41234 (41234 \times 10^{-5})</td>
<td>0.4 (4 \times 10^{-1})</td>
<td>0.4 (4 \times 10^{-1})</td>
</tr>
<tr>
<td>1</td>
<td>4.1234 (41234 \times 10^{-4})</td>
<td>4 (4 \times 10^{0})</td>
<td>4 (4 \times 10^{0})</td>
</tr>
<tr>
<td>1</td>
<td>41.234 (41234 \times 10^{-3})</td>
<td>4 (4 \times 10^{1})</td>
<td>4 (4 \times 10^{1})</td>
</tr>
<tr>
<td>1</td>
<td>412.34 (41234 \times 10^{-2})</td>
<td>4 (4 \times 10^{2})</td>
<td>4 (4 \times 10^{2})</td>
</tr>
<tr>
<td>2</td>
<td>0.491234 (491234 \times 10^{-6})</td>
<td>0.49 (49 \times 10^{-2})</td>
<td>0.49 (49 \times 10^{-2})</td>
</tr>
<tr>
<td>2</td>
<td>0.499876 (499876 \times 10^{-6})</td>
<td>0.49 (49 \times 10^{-2})</td>
<td>0.50 (50 \times 10^{-2})</td>
</tr>
<tr>
<td>2</td>
<td>0.999876 (999876 \times 10^{-6})</td>
<td>0.99 (99 \times 10^{-2})</td>
<td>1.0 (10 \times 10^{-1})</td>
</tr>
<tr>
<td>10</td>
<td>0.491234 (491234 \times 10^{-6})</td>
<td>0.491234 (491234 \times 10^{-6})</td>
<td>0.491234 (491234 \times 10^{-6})</td>
</tr>
<tr>
<td>10</td>
<td>999.999 (999999 \times 10^{-3})</td>
<td>999.999 (999999 \times 10^{-3})</td>
<td>999.999 (999999 \times 10^{-3})</td>
</tr>
<tr>
<td>10</td>
<td>9999999999999999 (9999999999999999 \times 10^{0})</td>
<td>9.999999999999999E+14 (9999999999999999 \times 10^{0})</td>
<td>1.0000000000E+15 (10000000000 \times 10^{0})</td>
</tr>
</tbody>
</table>

Figure 89. DFP Reround examples
### Programming Note

DFP **Reround** combined with **DFP Quantize** can be used to left justify a value (as needed by the frexp function). **DFP Reround** is the DFP value for which we want to left justify; \( f_{13} \) contains the reference significance value \( 0x0000000000000001 \); and \( r1 \) is the stack pointer, with free space for a doubleword at offset -8. This doubleword is used to transfer the biased exponents from the FPR to a GPR, for integer computation. The adjusted biased exponent (+ format precision - 1) is transferred back into an FPR so it can be inserted into the rerounded value. The adjusted rerounded value becomes the quantize reference value. The quantize instruction returns the left justified result in FRT.

\[
\text{drrnd } f1, f_{13}, FRB, 1 \# \text{reround 1 digit toward 0} \\
\text{dxex } f0, f1 \\
\text{stfd } f0, -8(r1) \\
\text{ldi } r11, -8(r1) \\
\text{addi } r11, r11, 15 \# \text{biased exp + precision - 1} \\
\text{ldi } r11, -8(r1) \\
\text{stfd } f0, -8(r1) \\
\text{diex } f1, f0, f1 \# \text{adjust exponent} \\
\text{dqua } FRT, f1, f0, 1 \# \text{quantize to adjusted exponent}
\]

### Actions for Reround when operand b in FRB\[p\] is

<table>
<thead>
<tr>
<th>( k \neq 0, k &lt; m )</th>
<th>( k \neq 0, k = m )</th>
<th>( k \neq 0 ) and ( k &gt; m ), or ( k = 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong></td>
<td><strong>fn</strong></td>
<td>( \infty )</td>
</tr>
<tr>
<td>( k \neq 0, k &lt; m )</td>
<td>-</td>
<td>( \text{RR}(b) ) or ( \text{VXCVI}: \text{T}(d\text{NaN}) )</td>
</tr>
<tr>
<td>( k \neq 0, k = m )</td>
<td>-</td>
<td>( \text{W}(b) )</td>
</tr>
<tr>
<td>( k = 0 ) and ( k &gt; m ), or ( k = 0 )</td>
<td>( \text{W}(b) )</td>
<td>( \text{W}(b) )</td>
</tr>
</tbody>
</table>

**Explanation:**

* The number of significant digits of the value 0 is considered to be zero for this instruction.
- Not applicable.

- **dINF**: Default infinity.
- **fn**: Finite nonzero numbers (includes both subnormal and normal numbers).
- **k**: Reference significance, which specifies the number of significant digits in the target operand.
- **m**: Number of significant digits in the operand in FRB\[p\].
- **P(x)**: The QNaN of operand x is propagated and placed in FRT\[p\].
- **RR(x)**: The value x is rounded to the form that has the specified number of significant digits.
  - If \( \text{RR}(x) \leq (10^{k-1} \times 10^{\text{max}}) \), then \( \text{RR}(x) \) is returned; otherwise an invalid-operation exception is recognized.
- **T(x)**: The value x is placed in FRT\[p\].
- **U(x)**: The SNaN of operand x is converted to the corresponding QNaN and placed in FRT\[p\].
- **VXCVI**: Floating-Point Invalid Operation (Invalid Conversion) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)
- **VXSNAN**: Floating-Point Invalid Operation (SNaN) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)
- **W(x)**: The value and the form of x is placed in FRT\[p\].

**Figure 90. Actions: Reround**
**DFP Round To FP Integer With Inexact Z23-form**

\[
\text{drntx} \quad R,FRT,FRB,RMC \quad (Rc=0) \\
\text{drntx.} \quad R,FRT,FRB,RMC \quad (Rc=1)
\]

The DFP operand in \( FRB[p] \) is rounded to a floating-point integer and placed into \( FRT[p] \). The sign of the result is the same as the sign of the operand in \( FRB[p] \). The ideal exponent is the larger value of zero and the exponent of the operand in \( FRB[p] \).

The rounding mode used is specified by \( RMC \). When the \( RMC\)-encoding-selection (R) bit is zero, \( RMC \) field contains the primary encoding; when the bit is one, the field contains the secondary encoding.

In addition to coercion of the converted value to fit the target format, the special rounding used by *Round To FP Integer* also coerces the target exponent to the ideal exponent.

When the operand in \( FRB[p] \) is a finite number and the exponent is less than zero, the operand is rounded to the result with an exponent of zero. When the exponent is greater than or equal to zero, the result is set to the numerical value and the form of the operand in \( FRB[p] \).

When the result differs in value from the operand in \( FRB[p] \), an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in \( FRB[p] \).

Figure 91 summarizes the actions for *Round To FP Integer With Inexact*. The table does not include the setting of \( FPRF \). \( FPRF \) is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

\[ \text{drntx[q][.:]} \text{ are treated as Floating-Point instructions in terms of resource availability.} \]

**Programming Note**

The **DFP Round To FP Integer With Inexact** and **DFP Round To FP Integer With Inexact Quad** instructions can be used to implement the decimal equivalent of the C99 rint function by specifying the primary \( RMC \) encoding for round according to DRN \((R=0, RMC=11)\). The specification for rint requires the inexact exception be raised if detected.

**Special Registers Altered:**

\[
\begin{array}{ccccccc}
\text{FPRF} & \text{FR} & \text{FI} & \text{FX} & \text{XX} & \text{VXSNAN} & \text{CR1}
\end{array}
\]

\( (\text{if } Rc=1) \)
### Table: Actions: Round to FP Integer With Inexact

| Operand b in FRB is | Is n not precise (n \(\neq b\)) | Inv.-Op. Exception Enabled | Inexact Exception Enabled | Is n Incremented (|n| > |b|) | Actions* |
|---------------------|----------------------------------|---------------------------|---------------------------|---------------------------------|----------|
| \(-\infty\)        | No\(^1\)                         |                           |                           |                                 | \(T(-d\text{INF}), \text{FI} \leftarrow 0, \text{FR} \leftarrow 0\) |
| F                  | No                               |                           |                           |                                 | \(W(n), \text{FI} \leftarrow 1, \text{FR} \leftarrow 0, \text{XX} \leftarrow 1\) |
| F                  | Yes                              |                           | No                        | No                              | \(W(n), \text{FI} \leftarrow 1, \text{FR} \leftarrow 1, \text{XX} \leftarrow 1\) |
| F                  | Yes                              |                           | No                        | No                              | \(W(n), \text{FI} \leftarrow 1, \text{FR} \leftarrow 1, \text{XX} \leftarrow 1\) |
| F                  | Yes                              |                           | Yes                       | No                              | \(W(n), \text{FI} \leftarrow 1, \text{FR} \leftarrow 1, \text{XX} \leftarrow 1, \text{TX}\) |
| \(+\infty\)        | No\(^1\)                         |                           |                           |                                 | \(T(+d\text{INF}), \text{FI} \leftarrow 0, \text{FR} \leftarrow 0\) |
| QNaN               | No\(^1\)                         |                           |                           |                                 | \(P(b), \text{FI} \leftarrow 0, \text{FR} \leftarrow 0\) |
| SNaN               | No\(^1\)                         |                           |                           |                                 | \(U(b), \text{FI} \leftarrow 0, \text{FR} \leftarrow 0, \text{VXSNAN} \leftarrow 1\) |
| SNaN               | No\(^1\)                         |                           |                           |                                 | \(VXSNAN \leftarrow 1, \text{TX}\) |

* Setting of XX and VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of \(\text{FI}\) and \(\text{FR}\) is part of the exception actions. (See the sections, "Inexact Exception" and "Invalid Operation Exception" for more details.)

- The actions do not depend on this condition.
- This condition is true by virtue of the state of some condition to the left of this column.

\(d\text{INF}\): Default infinity.
\(F\): All finite numbers, including zeros.
\(FI\): Floating-Point Fraction Inexact status bit.
\(FR\): Floating-Point Fraction Rounded status bit.
\(n\): The value derived when the source operand, b, is rounded to an integer using the special rounding for Round To FP Integer.
\(P(x)\): The QNaN of operand x is propagated and placed in \(FRT[p]\).
\(T(x)\): The value x is placed in \(FRT[p]\).
\(TV\): The system floating-point enabled exception error handler is invoked for the invalid operation exception if \(\text{FE}0\) and \(\text{FE}1\) are set to any mode other than the ignore-exception mode.
\(TX\): The system floating-point enabled exception error handler is invoked for the inexact exception if \(\text{FE}0\) and \(\text{FE}1\) are set to any mode other than the ignore-exception mode.
\(U(x)\): The SNaN of operand x is converted to the corresponding QNaN and placed in \(FRT[p]\).
\(W(x)\): The value x in the form of zero exponent or the source exponent is placed in \(FRT[p]\).

**Explanation:**

**Figure 91. Actions: Round to FP Integer With Inexact**
DFP Round To FP Integer Without Inexact
Z23-form

drintn R,FRT,FRB,RMC (Rc=0)
drintn R,FRT,FRB,RMC (Rc=1)

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>///</th>
<th>R</th>
<th>FRB</th>
<th>RMC</th>
<th>227</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>23</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

DFP Round To FP Integer Without Inexact Quad Z23-form

drintnq R,FRTp,FRBp,RMC (Rc=0)
drintnq R,FRTp,FRBp,RMC (Rc=1)

<table>
<thead>
<tr>
<th>63</th>
<th>FRTp</th>
<th>///</th>
<th>R</th>
<th>FRBp</th>
<th>RMC</th>
<th>227</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>23</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

This operation is the same as the Round To FP Integer With Inexact operation, except that this operation does not recognize an inexact exception.

Figure 92 summarizes the actions for Round To FP Integer Without Inexact. The table does not include the setting of FPRF. FPRF is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

drintn[q][.] are treated as Floating-Point instructions in terms of resource availability.

Special Registers Altered:
FPRF FR (set to 0) F1 (set to 0)
FX
VXSNAN
CR1 (if Rc=1)

Programming Note

The DFP Round To FP Integer Without Inexact and DFP Round To FP Integer Without Inexact Quad instructions can be used to implement decimal equivalents of several C99 rounding functions by specifying the appropriate R and RMC field values.

<table>
<thead>
<tr>
<th>Function</th>
<th>R</th>
<th>RMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ceil</td>
<td>1</td>
<td>0b00</td>
</tr>
<tr>
<td>Floor</td>
<td>1</td>
<td>0b01</td>
</tr>
<tr>
<td>Nearbyint</td>
<td>0</td>
<td>0b11</td>
</tr>
<tr>
<td>Round</td>
<td>0</td>
<td>0b10</td>
</tr>
<tr>
<td>Trunc</td>
<td>0</td>
<td>0b01</td>
</tr>
</tbody>
</table>

Note that nearbyint is similar to the rint function but without raising the inexact exception. Similarly ceil, floor, round, and trunc do not require the inexact exception.
<table>
<thead>
<tr>
<th>Operand b in FRB is</th>
<th>Inv.-Op. Exception Enabled</th>
<th>Actions*</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>-</td>
<td>[T(-d_{\text{Inf}}), F_{1} ← 0, FR ← 0]</td>
</tr>
<tr>
<td>(\text{F})</td>
<td>-</td>
<td>[W(n), F_{1} ← 0, FR ← 0]</td>
</tr>
<tr>
<td>+∞</td>
<td>-</td>
<td>[T(+d_{\text{Inf}}), F_{1} ← 0, FR ← 0]</td>
</tr>
<tr>
<td>QNaN</td>
<td>-</td>
<td>[P(b), F_{1} ← 0, FR ← 0]</td>
</tr>
<tr>
<td>SNaN</td>
<td>No</td>
<td>[U(b), F_{1} ← 0, FR ← 0, VXSNAN ← 1]</td>
</tr>
<tr>
<td>SNaN</td>
<td>Yes</td>
<td>[VXSNAN ← 1, TV]</td>
</tr>
</tbody>
</table>

Explanation:
- Setting of \(VXSNAN\) is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of \(F_{1}\) and \(FR\) bits is part of the exception actions. (See the sections, "Invalid Operation Exception" for more details.)
- The actions do not depend on this condition.

\(d_{\text{Inf}}\) Default infinity.
\(F\) All finite numbers, including zeros.
\(F_{1}\) Floating-Point Fraction Inexact status bit.
\(FR\) Floating-Point Fraction Rounded status bit.
\(n\) The value derived when the source operand, \(b\), is rounded to an integer using the special rounding for Round To FP Integer.
\(P(x)\) The QNaN of operand \(x\) is propagated and placed in \(FRT[p]\).
\(T(x)\) The value \(x\) is placed in \(FRT[p]\).
\(TV\) The system floating-point enabled exception error handler is invoked for the invalid-operation exception if \(FE0\) and \(FE1\) are set to any mode other than the ignore-exception mode.
\(U(x)\) The SNaN of operand \(x\) is converted to the corresponding QNaN and placed in \(FRT[p]\).
\(W(x)\) The value \(x\) in the form of zero exponent or the source exponent is placed in \(FRT[p]\).

Figure 92. Actions: Round to FP Integer Without Inexact
5.6.5 DFP Conversion Instructions

The DFP conversion instructions consist of data-format conversion instructions and data-type conversion instructions. They are all X-form instructions and employ the record bit (Rc).

5.6.5.1 DFP Data-Format Conversion Instructions

The data-format conversion instructions consist of Convert To DFP Long, Convert To DFP Extended, Round To DFP Short, and Round To DFP Long. Figure 93 summarizes the actions for these instructions.

### Programming Note

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP Load Floating-Point Single and Store Floating-Point Single functions, the widening is performed by loading the DFP short value with a Load Floating as Integer Word Indexed followed by a DFP Convert to DFP Long, and narrowing is performed by a DFP Round to DFP Short followed by a Store Floating-Point as Integer Word Indexed. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.

---

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Actions when operand b in FRB[p] is</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>F</strong></td>
</tr>
<tr>
<td>Convert To DFP Long</td>
<td>T(b) 1</td>
</tr>
<tr>
<td>Convert To DFP Extended</td>
<td>T(b) 1</td>
</tr>
<tr>
<td>Round To DFP Short</td>
<td>R(b) 1</td>
</tr>
<tr>
<td>Round To DFP Long</td>
<td>R(b) 1</td>
</tr>
</tbody>
</table>

**Explanation:**

1. The ideal exponent is the exponent of the source operand.
2. Bits 5:N-1 of the N-bit combination field are set to zero.
3. Bit 5 of the N-bit combination field is set to one. Bits 6:N-1 of the combination field are set to zero.
4. The trailing significand field is padded on the left with zeros.
5. Leftmost digits in the trailing significand field are removed.
6. Default infinity.
7. All finite numbers, including zeros.
8. The special symbol in operand x is propagated into FRT[p].
9. The value x is rounded to the target-format precision; see Section 5.5.11
10. The value x is placed in FRT[p].
11. The SNaN of operand x is converted to the corresponding QNaN.
12. Floating-Point Invalid Operation (SNaN) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 199.)

Figure 93. Actions: Data-Format Conversion Instructions
**DFP Convert To DFP Long X-form**

\[
\begin{array}{cccccc}
\text{Rc} & \text{0} & \text{6} & \text{11} & \text{16} & \text{21} & \text{31} \\
\text{FRT} & \text{FRB} & \text{258} & \\
\text{0} & \text{32:63} & \\
\end{array}
\]

The DFP short operand in bits 32:63 of FRB is converted to DFP long format and the converted result is placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the source operand.

If the operand in FRB is an SNaN, it is converted to an SNaN in DFP long format and does not cause an invalid-operation exception.

\[
\text{dctdp} \quad \text{FRT,FRB} \quad (\text{Rc}=0) \\
\text{dctdp} \quad \text{FRT,FRB} \quad (\text{Rc}=1)
\]

**DFP Convert To DFP Extended X-form**

\[
\begin{array}{cccccc}
\text{Rc} & \text{0} & \text{6} & \text{11} & \text{16} & \text{21} & \text{31} \\
\text{FRT} & \text{FRB} & \text{258} & \\
\text{0} & \text{63} & \text{11} & \text{16} & \text{21} & \text{31} \\
\end{array}
\]

The DFP long operand in the FRB is converted to DFP extended format and placed into FRTp. The sign of the result is the same as the sign of the operand in FRB. The ideal exponent is the exponent of the operand in FRB.

If the operand in FRB is an SNaN, an invalid-operation exception is recognized. If the exception is disabled, the SNaN is converted to the corresponding QNaN in DFP extended format.

\[
\text{dctpq} \quad \text{FRTp,FRB} \quad (\text{Rc}=0) \\
\text{dctpq} \quad \text{FRTp,FRB} \quad (\text{Rc}=1)
\]

**Programming Note**

Note that DFP short format is a storage-only format, Therefore, conversion of a short SNaN to long format will not cause an exception and the SNaN is preserved. Subsequent operation on that SNaN in long format will cause an exception.
**DFP Round To DFP Short X-form**

drsp  FRT,FRB  \((Rc=0)\)
drsp. FRT,FRB  \((Rc=1)\)

The DFP long operand in \(FRB\) is converted and rounded to DFP short format. The DFP short value is extended on the left with zeros to form a 64-bit entity and placed into \(FRT\). The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the source operand.

If the operand in \(FRB\) is an SNaN, it is converted to an SNaN in DFP short format and does not cause an invalid-operation exception.

Normally, the result is in the format and length of the target. However, when an overflow or underflow exception occurs and if the exception is enabled, the operation is completed by producing a wrapped rounded result in the same format and length as the source but rounded to the target-format precision.

**Programming Note**

Note that DFP short format is a storage-only format, therefore, conversion of a long SNaN to short format will not cause an exception. Converting a long format SNaN to short format is an implied move operation.

**Special Registers Altered:**

\[
\begin{array}{cccccc}
FPRF & FR & FI & FX & OX & UX & XX \\
CR1 & & & & & & (if \(Rc=1\))
\end{array}
\]

**drsp[.]** are treated as Floating-Point instructions in terms of resource availability.

**DFP Round To DFP Long X-form**

drdpq  FRTp,FRBp  \((Rc=0)\)
drdpq. FRTp,FRBp  \((Rc=1)\)

The DFP extended operand in \(FRBp\) is converted and rounded to DFP long format. The result concatenated with 64 0s is placed in \(FRTp\). The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the operand in \(FRBp\).

If the operand in \(FRBp\) is an SNaN, an invalid-operation exception is recognized. If the exception is disabled, the SNaN is converted to the corresponding QNaN in DFP long format.

Normally, the result is in the format and length of the target. However, when an overflow or underflow exception occurs and if the exception is enabled, the operation is completed by producing a wrapped rounded result in the same format and length as the source but rounded to the target-format precision.

**Programming Note**

Note that DFP Round to DFP Long, while producing a result in DFP long format, actually targets a register pair, writing 64 0s in \(FRTp+1\).

**Special Registers Altered:**

\[
\begin{array}{cccccc}
FPRF & FR & FI & FX & OX & UX & XX \\
VXSNAN & & & & & & (if \(Rc=1\))
\end{array}
\]

**drdpq[.]** are treated as Floating-Point instructions in terms of resource availability.
5.6.5.2 DFP Data-Type Conversion Instructions

The DFP data-type conversion instructions are used to convert data type between DFP and fixed.

The data-type conversion instructions consist of Convert From Fixed and Convert To Fixed.

### DFP Convert From Fixed X-form

<table>
<thead>
<tr>
<th>dcffix</th>
<th>FRT,FRB (Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcffix</td>
<td>FRT,FRB (Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>//</th>
<th>FRB</th>
<th>802</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

The 64-bit signed binary integer in FRB is converted and rounded to a DFP Long value and placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

FPRF is set to the class and sign of the result.

**dcffix[]** are treated as Floating-Point instructions in terms of resource availability.

**Special Registers Altered:**

- FPRF FR FI
- FX XX
- CR1 (if Rc=1)

### DFP Convert From Fixed Quad X-form

<table>
<thead>
<tr>
<th>dcffixq</th>
<th>FRTp,FRB (Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcffixq</td>
<td>FRTp,FRB (Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>63</th>
<th>FRTp</th>
<th>//</th>
<th>FRB</th>
<th>802</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

The 64-bit signed binary integer in FRB is converted and rounded to a DFP Extended value and placed into FRTp. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

FPRF is set to the class and sign of the result.

**dcffixq[]** are treated as Floating-Point instructions in terms of resource availability.

**Special Registers Altered:**

- FPRF FR (undefined) FI (undefined)
- CR1 (if Rc=1)
**DFP Convert From Fixed Quadword Quad X-form**

\[ \text{dcffixqq} \rightarrow \text{FRTp, VRB} \]

<table>
<thead>
<tr>
<th>63</th>
<th>FRTp</th>
<th>VRB</th>
<th>994</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

The 128-bit signed binary integer in VRB is converted and rounded to a DFP Extended value and placed into FRTp. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

FPRF is set to the class and sign of the result.

\( \text{dcffixqq} \) is treated as a Floating-Point and a Vector instruction in terms of resource availability.

**Special Registers Altered:**

FPRF FR FI FX XX

**VSR Data Layout for dcffixqq**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>VSR[FRTp].dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VSR[FRTp+1].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

VSR Data Layout for dcffixqq
**DFP Convert To Fixed X-form**

<table>
<thead>
<tr>
<th>dctfix</th>
<th>FRT,FRB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dctfix.</td>
<td>FRT,FRB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

The DFP operand in FRB[p] is rounded to an integer value and is placed into FRT in the 64-bit signed binary integer format. The sign of the result is the same as the sign of the source operand, except when the source operand is a NaN or a zero.

Figure 94 summarizes the actions for Convert To Fixed.

**dfctfix[q][]** are treated as Floating-Point instructions in terms of resource availability.

**Special Registers Altered:**
- FPRF (undefined)
- FR FI
- FX XX
- VXSNAN VXCVI
- CR1

**Programming Note**

It is recommended that software pre-round the operand to a floating-point integral using **drintx[q]** or **drintn[q]** if a rounding mode other than the current rounding mode specified by DRN is needed. Saving, modifying and restoring the FPSCR just to temporarily change the rounding mode is less efficient than just employing drintx[p] or drint[p] which override the current rounding mode using an immediate control field.

For example if the desired function rounding is Round to Nearest, Ties away from 0 but the default rounding (from DRN) is Round to Nearest, Ties to Even then following is preferred.

```
drintn  0,f1,f1,2
dctfix  f1,f1
```

**VSR Data Layout for dctfixqq**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[FRBp].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VSR[FRBp+1].dword[0]</td>
<td>unused</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>290</td>
<td>Rq</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>290</td>
</tr>
<tr>
<td>Rq</td>
<td>0</td>
</tr>
</tbody>
</table>

**DFP Convert To Fixed Quad X-form**

<table>
<thead>
<tr>
<th>dctfixq</th>
<th>FRT,FRBp</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dctfixq.</td>
<td>FRT,FRBp</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

The DFP operand in FRB[p] is rounded to an integer value and is placed into VRT in the 128-bit signed binary integer format. The sign of the result is the same as the sign of the source operand, except when the source operand is a NaN or a zero.

Figure 96 summarizes the actions for Convert To Fixed.

**dctfixqq** is treated as a Floating-Point and a Vector instruction in terms of resource availability.

**Special Registers Altered:**
- FPRF (undefined)
- FR FI
- FX VXSNAN VXCVI
- XX
Table: Actions: Convert To Fixed

| Operand b in FRB[p] is | q is | Is n not precise (n ≠ b) | Inv.-Op. Except. Enabled | Inexact Except. Enabled | Is n Incremented (|n| > |b|) | Actions * |
|------------------------|------|--------------------------|--------------------------|------------------------|-----------------------------|----------|
| -∞ ≤ b < MN            | < MN | No                       | -                         | -                      | -                           | T(MN), FI ← 0, FR ← 0, VXCVI ← 1 |
| -∞ ≤ b < MN            | < MN | Yes                       | -                         | -                      | -                           | VXCVI ← 1, TV                |
| -∞ < b < MN            | = MN | No                        | -                         | -                      | -                           | T(MN), FI ← 1, FR ← 0, XX ← 1 |
| -∞ < b < MN            | = MN | Yes                       | -                         | -                      | -                           | T(MN), FI ← 1, FR ← 0, XX ← 1, TX |
| MN ≤ b < 0             | No   | -                         | -                         | -                      | -                           | T(n), FI ← 0, FR ← 0          |
| MN ≤ b < 0             | Yes  | -                         | No                        | -                      | No                          | T(n), FI ← 1, FR ← 0, XX ← 1 |
| MN ≤ b < 0             | Yes  | -                         | Yes                       | No                     | Yes                         | T(n), FI ← 1, FR ← 1, XX ← 1 |
| MN ≤ b < 0             | Yes  | -                         | Yes                       | Yes                    | Yes                         | T(n), FI ← 1, FR ← 1, XX ← 1, TX |
| ±0                     | No   | -                         | -                         | -                      | -                           | T(0), FI ← 0, FR ← 0          |
| 0 < b ≤ MP             | No   | -                         | -                         | -                      | -                           | T(n), FI ← 0, FR ← 0          |
| 0 < b ≤ MP             | Yes  | -                         | No                        | -                      | No                          | T(n), FI ← 1, FR ← 0, XX ← 1 |
| 0 < b ≤ MP             | Yes  | -                         | No                        | Yes                    | Yes                         | T(n), FI ← 1, FR ← 1, XX ← 1 |
| 0 < b ≤ MP             | Yes  | -                         | Yes                       | Yes                    | Yes                         | T(n), FI ← 1, FR ← 1, XX ← 1, TX |
| MP < b < +∞            | = MP | -                         | -                         | No                     | -                           | T(MP), FI ← 0, FR ← 0, XX ← 1 |
| MP < b < +∞            | = MP | -                         | -                         | Yes                    | -                           | T(MP), FI ← 1, FR ← 0, XX ← 1, TX |
| MP < b ≤ +∞            | > MP | -                         | No                        | -                      | -                           | T(p), FI ← 0, FR ← 0, VXCVI ← 1 |
| MP < b ≤ +∞            | > MP | Yes                       | -                         | -                      | -                           | VXCVI ← 1, TV                 |
| QNaN                   | No   | -                         | -                         | -                      | -                           | T(MN), FI ← 0, FR ← 0, VXCVI ← 1 |
| QNaN                   | Yes  | -                         | -                         | -                      | -                           | VXCVI ← 1, TV                 |
| SNaN                   | No   | -                         | -                         | -                      | -                           | VXCVI ← 1, VXSNAN ← 1         |
| SNaN                   | Yes  | -                         | -                         | -                      | -                           | VXCVI ← 1, VXSNAN ← 1, TV     |

Explanation:

* Setting of XX, VXCVI, and VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and FR bits is part of the exception actions. (See the sections, “Inexact Exception” and “Invalid Operation Exception” for more details.)

- The actions do not depend on this condition.

FI Floating-Point Fraction Inexact status bit.
FR Floating-Point Fraction Rounded status bit.
MN Maximum negative number representable by the 64-bit binary integer format
MP Maximum positive number representable by the 64-bit binary integer format
n The value q converted to a fixed-point result.
q The value derived when the source value b is rounded to an integer using the specified rounding mode
T(x) The value x is placed in FRT[p].
TV The system floating-point enabled exception error handler is invoked for the invalid-operation exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.
TX The system floating-point enabled exception error handler is invoked for the inexact exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.
VXCVI Floating-Point Invalid Operation (Invalid Conversion) exception status bit.
VXSNAN Floating-Point Invalid Operation (SNaN) exception status bit.
XX Floating-Point Inexact exception status bit.

Figure 94. Actions: Convert To Fixed
5.6.6 DFP Format Instructions

The DFP format instructions are used to compose or decompose a DFP operand. A source operand of SNaN does not cause an invalid-operation exception. All format instructions employ the record bit (Rc).

DFP Decode DPD To BCD X-form

<table>
<thead>
<tr>
<th>ddedpd</th>
<th>SP, FRT, FRB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddedpd.</td>
<td>SP, FRT, FRB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

DFP Decode DPD To BCD Quad X-form

<table>
<thead>
<tr>
<th>ddedpdq</th>
<th>SP, FRTp, FRBp</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddedpdq.</td>
<td>SP, FRTp, FRBp</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

A portion of the significand of the DFP operand in FRB[p] is converted to a signed or unsigned BCD number depending on the SP field. For infinity and NaN, the significand is considered to be the contents in the trailing significand field padded on the left by a zero digit.

SP0 = 0 (unsigned conversion)

The rightmost 16 digits of the significand (32 digits for ddedpdq) is converted to an unsigned BCD number and the result is placed into FRT[p].

SP0 = 1 (signed conversion)

The rightmost 15 digits of the significand (31 digits for ddedpdq) is converted to a signed BCD number with the same sign as the DFP operand, and the result is placed into FRT[p]. If the DFP operand is negative, the sign is encoded as 0b1101. If the DFP operand is positive, SP1 indicates which preferred plus sign encoding is used. If SP1 = 0, the plus sign is encoded as 0b1100 (the option-1 preferred sign code), otherwise the plus sign is encoded as 0b1111 (the option-2 preferred sign code).

ddedpdq[p][] are treated as Floating-Point instructions in terms of resource availability.

Special Registers Altered:

CR1 \((\text{if } Rc=1)\)

DFP Encode BCD To DPD X-form

<table>
<thead>
<tr>
<th>denbcd</th>
<th>S, FRT, FRB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>denbcd.</td>
<td>S, FRT, FRB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

DFP Encode BCD To DPD Quad X-form

<table>
<thead>
<tr>
<th>denbcdq</th>
<th>S, FRTp, FRBp</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>denbcdq.</td>
<td>S, FRTp, FRBp</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

The signed or unsigned BCD operand, depending on the S field, in FRB[p] is converted to a DFP number. The ideal exponent is zero.

S = 0 (unsigned BCD operand)

The unsigned BCD operand in FRB[p] is converted to a positive DFP number of the same magnitude and the result is placed into FRT[p].

S = 1 (signed BCD operand)

The signed BCD operand in FRB[p] is converted to the corresponding DFP number and the result is placed into FRT[p].

If an invalid BCD digit or sign code is detected in the source operand, an invalid-operation exception (VXCVI) occurs.

FPRF is set to the class and sign of the result, except for Invalid Operation Exception when VE=1.

denbcdq[p][] are treated as Floating-Point instructions in terms of resource availability.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXCVI
CR1 \((\text{if } Rc=1)\)
**DFP Extract Biased Exponent X-form**

\[
dxex \rightarrow \text{FRT,FRB} \quad (Rc=0) \\
dxex. \rightarrow \text{FRT,FRB} \quad (Rc=1)
\]

**DFP Extract Biased Exponent Quad X-form**

\[
dxexq \rightarrow \text{FRT,FRBp} \quad (Rc=0) \\
dxexq. \rightarrow \text{FRT,FRBp} \quad (Rc=1)
\]

The biased exponent of the operand in \(\text{FRB}[p]\) is extracted and placed into \(\text{FRT}\) in the 64-bit signed binary integer format. When the operand in \(\text{FRB}\) is an infinity, QNaN, or SNaN, a special code is returned.

\(dxex[q][]\) are treated as Floating-Point instructions in terms of resource availability.

**Special Registers Altered:**

- CR1 
  (if \(Rc=1\))

---

**Programming Note**

The exponent bias value is 101 for DFP Short, 398 for DFP Long, and 6176 for DFP Extended.

---

**DFP Insert Biased Exponent X-form**

\[
diex \rightarrow \text{FRT,FRA,FRB} \quad (Rc=0) \\
diex. \rightarrow \text{FRT,FRA,FRB} \quad (Rc=1)
\]

**DFP Insert Biased Exponent Quad X-form**

\[
diexq \rightarrow \text{FRTp,FRA,FRBp} \\
diexq. \rightarrow \text{FRTp,FRA,FRBp} \quad (Rc=1)
\]

Let \(a\) be the value of the 64-bit signed binary integer in \(\text{FRA}\).

\[
\begin{align*}
\text{Operand} & \quad \text{Result} \\
\text{Finite Number} & \quad \text{biased exponent value} \\
\text{Infinity} & \quad -1 \\
\text{QNaN} & \quad -2 \\
\text{SNaN} & \quad -3 \\
\end{align*}
\]

When \(0 \leq a \leq \text{MBE}\), \(a\) is the biased target exponent that is combined with the sign bit and the significand value of the DFP operand in \(\text{FRB}[p]\) to form the DFP result in \(\text{FRT}[p]\). The ideal exponent is the specified target exponent.

When \(a\) specifies a special code (\(a < 0\) or \(a > \text{MBE}\)), an infinity, QNaN, or SNaN is formed in \(\text{FRT}[p]\) with the trailing significand field containing the value from the trailing significand field of the source operand in \(\text{FRB}[p]\), and with an N-bit combination field set as follows.

- For an Infinity result,
  - the leftmost 5 bits are set to \(0b11110\), and
  - the rightmost N-5 bits are set to zero.
- For a QNaN result,
  - the leftmost 5 bits are set to \(0b11111\),
  - bit 5 is set to zero, and
  - the rightmost N-5 bits are set to zero.
- For an SNaN result,
  - the leftmost 5 bits are set to \(0b11111\),
  - bit 5 is set to one, and
  - the rightmost N-5 bits are set to zero.

\(dxex[q][]\) are treated as Floating-Point instructions in terms of resource availability.

**Special Registers Altered:**

- CR1 
  (if \(Rc=1\))

---

**Programming Note**

The exponent bias value is 101 for DFP Short, 398 for DFP Long, and 6176 for DFP Extended.
Operand a in \( FRA[p] \) specifies

<table>
<thead>
<tr>
<th>Actions for Insert Biased Exponent when operand b in FRB[p] specifies</th>
<th>F</th>
<th>( \infty )</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F )</td>
<td>( F ), ( Rb )</td>
<td>( Z, Rb )</td>
<td>( Z, Rb )</td>
<td>( Z, Rb )</td>
</tr>
<tr>
<td>( \infty )</td>
<td>( I, Rb )</td>
<td>( I, Rb )</td>
<td>( I, Rb )</td>
<td>( I, Rb )</td>
</tr>
<tr>
<td>QNaN</td>
<td>( Q, Rb )</td>
<td>( Q, Rb )</td>
<td>( Q, Rb )</td>
<td>( Q, Rb )</td>
</tr>
<tr>
<td>SNaN</td>
<td>( S, Rb )</td>
<td>( S, Rb )</td>
<td>( S, Rb )</td>
<td>( S, Rb )</td>
</tr>
</tbody>
</table>

Explanation:

- **F**: All finite numbers, including zeros
- **I**: The combination field in \( FRT[p] \) is set to indicate a default Infinity.
- **N**: The combination field in \( FRT[p] \) is set to the specified biased exponent in \( FRA \) and the leftmost significand digit in \( FRB[p] \).
- **Q**: The combination field in \( FRT[p] \) is set to indicate a default QNaN.
- **S**: The combination field in \( FRT[p] \) is set to indicate a default SNaN.
- **Z**: The combination field in \( FRT[p] \) is set to indicate the specific biased exponent in \( FRA \) and a leftmost coefficient digit of zero.
- **Rb**: The contents of the trailing significand field in \( FRB[p] \) are reencoded using preferred DPD encodings and the reencoded result is placed in the same field in \( FRT[p] \). The sign bit of \( FRB[p] \) is copied into the sign bit in \( FRT[p] \).

**Figure 95. Actions: Insert Biased Exponent**
### DFP Shift Significand Left Immediate Z22-form

<table>
<thead>
<tr>
<th>Code</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>dscl</td>
<td>FRT,FRA,SH</td>
</tr>
<tr>
<td>dscl.</td>
<td>FRT,FRA,SH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>59</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>66</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### DFP Shift Significand Left Immediate Quad Z22-form

<table>
<thead>
<tr>
<th>Code</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsclq</td>
<td>FRTp,FRAp,SH</td>
</tr>
<tr>
<td>dsclq.</td>
<td>FRTp,FRAp,SH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>66</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

The significand of the DFP operand in FRA[p] is shifted left \( SH \) digits. For a NaN or infinity, all significand digits are in the trailing significand field. \( SH \) is a 6-bit unsigned binary integer. Digits shifted out of the leftmost digit are lost. Zeros are supplied to the vacated positions on the right. The result is placed into FRT[p]. The sign of the result is the same as the sign of the source operand in FRA[p].

If the source operand in FRA[p] is a finite number, the exponent of the result is the same as the exponent of the source operand.

For an Infinity, QNaN or SNaN result, the target format’s N-bit combination field is set as follows.

- For an Infinity result,
  - the leftmost 5 bits are set to 0b11110, and
  - the rightmost N-5 bits are set to zero.

- For a QNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to zero, and
  - the rightmost N-6 bits are set to zero.

- For an SNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to one, and
  - the rightmost N-6 bits are set to zero.

**dscl[q][.]** are treated as *Floating-Point* instructions in terms of resource availability.

### Special Registers Altered:

**CR1** (if Rc = 1)

### DFP Shift Significand Right Immediate Z22-form

<table>
<thead>
<tr>
<th>Code</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>dscr</td>
<td>FRT,FRA,SH</td>
</tr>
<tr>
<td>dscr.</td>
<td>FRT,FRA,SH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>59</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>98</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

The significand of the DFP operand in FRA[p] is shifted right \( SH \) digits. For a NaN or infinity, all significand digits are in the trailing significand field. \( SH \) is a 6-bit unsigned binary integer. Digits shifted out of the units digit are lost. Zeros are supplied to the vacated positions on the left. The result is placed into FRT[p]. The sign of the result is the same as the sign of the source operand in FRA[p].

If the source operand in FRA[p] is a finite number, the exponent of the result is the same as the exponent of the source operand.

For an Infinity, QNaN or SNaN result, the target format’s N-bit combination field is set as follows.

- For an Infinity result,
  - the leftmost 5 bits are set to 0b11110, and
  - the rightmost N-5 bits are set to zero.

- For a QNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to zero, and
  - the rightmost N-6 bits are set to zero.

- For an SNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to one, and
  - the rightmost N-6 bits are set to zero.

**dscr[q][.]** are treated as *Floating-Point* instructions in terms of resource availability.

### Special Registers Altered:

**CR1** (if Rc = 1)
### 5.6.7 DFP Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Full Name</th>
<th>FORM</th>
<th>Operands</th>
<th>SNaN</th>
<th>Encoding</th>
<th>FP Exception</th>
<th>FRFI</th>
<th>IE</th>
<th>oc</th>
<th>cc</th>
</tr>
</thead>
<tbody>
<tr>
<td>dadd</td>
<td>DFP Add</td>
<td>X</td>
<td>FRT, FRA, FRB</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>daddq</td>
<td>DFP Add Quad</td>
<td>X</td>
<td>FRTp, FRAp, FRBp</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dsub</td>
<td>DFP Subtract</td>
<td>X</td>
<td>FRT, FRA, FRB</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dsubq</td>
<td>DFP Subtract Quad</td>
<td>X</td>
<td>FRTp, FRAp, FRBp</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmul</td>
<td>DFP Multiply</td>
<td>X</td>
<td>FRT, FRA, FRB</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmulq</td>
<td>DFP Multiply Quad</td>
<td>X</td>
<td>FRTp, FRAp, FRBp</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ddv</td>
<td>DFP Divide</td>
<td>X</td>
<td>FRT, FRA, FRB</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>Z O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ddvq</td>
<td>DFP Divide Quad</td>
<td>X</td>
<td>FRTp, FRAp, FRBp</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>Z O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dcmpo</td>
<td>DFP Compare Ordered</td>
<td>X</td>
<td>BF, FRA, FRB</td>
<td>Y - -</td>
<td>N Y V</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dcmpoq</td>
<td>DFP Compare Ordered Quad</td>
<td>X</td>
<td>BF, FRAp, FRBp</td>
<td>Y - -</td>
<td>N Y V</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dcmpu</td>
<td>DFP Compare Unordered</td>
<td>X</td>
<td>BF, FRA, FRB</td>
<td>Y - -</td>
<td>N Y V</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dcmpuq</td>
<td>DFP Compare Unordered Quad</td>
<td>X</td>
<td>BF, FRAp, FRBp</td>
<td>Y - -</td>
<td>N Y V</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtstdc</td>
<td>DFP Test Data Class</td>
<td>Z22</td>
<td>BF, FRA, DCM</td>
<td>N - -</td>
<td>N Y¹</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtstdcq</td>
<td>DFP Test Data Class Quad</td>
<td>Z22</td>
<td>BF, FRAp, DCM</td>
<td>N - -</td>
<td>N Y¹</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtstdg</td>
<td>DFP Test Data Group</td>
<td>Z22</td>
<td>BF, FRA, DGM</td>
<td>N - -</td>
<td>N Y¹</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtstdgq</td>
<td>DFP Test Data Group Quad</td>
<td>Z22</td>
<td>BF, FRAp, DGM</td>
<td>N - -</td>
<td>N Y¹</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtstex</td>
<td>DFP Test Exponent</td>
<td>X</td>
<td>BF, FRA, FRB</td>
<td>N - -</td>
<td>N Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtstexq</td>
<td>DFP Test Exponent Quad</td>
<td>X</td>
<td>BF, FRAp, FRBp</td>
<td>N - -</td>
<td>N Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtstsf</td>
<td>DFP Test Significance</td>
<td>X</td>
<td>BF, FRA(FIX), FRB</td>
<td>N - -</td>
<td>N Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtstsfq</td>
<td>DFP Test Significance Quad</td>
<td>X</td>
<td>BF, FRA(FIX), FRBp</td>
<td>N - -</td>
<td>N Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dquai</td>
<td>DFP Quantize Immediate</td>
<td>Z23</td>
<td>TE, FRT, FRB, RMC</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>X Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dquiaq</td>
<td>DFP Quantize Immediate Quad</td>
<td>Z23</td>
<td>TE, FRTp, FRBp, RMC</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>X Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dqua</td>
<td>DFP Quantize</td>
<td>Z23</td>
<td>FRT, FRA, FRB, RMC</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>X Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dquag</td>
<td>DFP Quantize Quad</td>
<td>Z23</td>
<td>FRTp, FRAp, FRBp, RMC</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>X Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drmd</td>
<td>DFP Round</td>
<td>Z23</td>
<td>FRT, FRA(FIX), FRB, RMC</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>X Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drmdq</td>
<td>DFP Round Quard</td>
<td>Z23</td>
<td>FRTp, FRA(FIX), FRBp, RMC</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>X Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drintx</td>
<td>DFP Round To FP Integer With Inexact</td>
<td>Z23</td>
<td>R, FRT, FRB, RMC</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>X Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drintxq</td>
<td>DFP Round To FP Integer With Inexact Quad</td>
<td>Z23</td>
<td>R, FRTp, FRBp, RMC</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>X Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drintn</td>
<td>DFP Round To FP Integer Without Inexact</td>
<td>Z23</td>
<td>R, FRT, FRB, RMC</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>y²</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>drintnq</td>
<td>DFP Round To FP Integer Without Inexact Quad</td>
<td>Z23</td>
<td>R, FRTp, FRBp, RMC</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>y²</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dctdp</td>
<td>DFP Convert To DFP Long</td>
<td>X</td>
<td>FRT, FRB (DFP Short)</td>
<td>N Y RE</td>
<td>Y y²</td>
<td>U Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dctdpq</td>
<td>DFP Convert To DFP Extended</td>
<td>X</td>
<td>FRTp, FRB</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>y²</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>drsp</td>
<td>DFP Round To DFP Short</td>
<td>X</td>
<td>FRT (DFP Short), FRB</td>
<td>N Y RE</td>
<td>Y y²</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drdpq</td>
<td>DFP Round To DFP Long</td>
<td>X</td>
<td>FRTp, FRBp</td>
<td>Y N RE</td>
<td>Y Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 96. Decimal Floating-Point Instructions Summary
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Full Name</th>
<th>FORM</th>
<th>Operands</th>
<th>SNaN</th>
<th>Encoding</th>
<th>FPRF</th>
<th>FP Exception</th>
<th>RMC</th>
<th>S</th>
<th>IE</th>
<th>RO</th>
</tr>
</thead>
<tbody>
<tr>
<td>dccfixq</td>
<td>DFP Convert From Fixed Quad</td>
<td>X</td>
<td>FRTp, FRB (FIX)</td>
<td>-</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>U</td>
<td>Y</td>
</tr>
<tr>
<td>dccfix</td>
<td>DFP Convert To Fixed</td>
<td>X</td>
<td>FRT (FIX), FRB</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>U</td>
<td>U</td>
<td>V</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>dctxfixq</td>
<td>DFP Convert To Fixed Quad</td>
<td>X</td>
<td>FRT (FIX), FRBp</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>U</td>
<td>U</td>
<td>V</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>dctxfixqq</td>
<td>DFP Convert From Fixed Quadword</td>
<td>x</td>
<td>FRTp, VRB (FIX)</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>U</td>
<td>Y</td>
</tr>
<tr>
<td>dctxfixq</td>
<td>DFP Convert To Fixed Quadword</td>
<td>x</td>
<td>VRT (FIX), FRBp</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>U</td>
<td>U</td>
<td>V</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>ddedpd</td>
<td>DFP Decode DPD To BCD</td>
<td>X</td>
<td>SP, FRT (BCD), FRB</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>ddedpdq</td>
<td>DFP Decode DPD To BCD Quad</td>
<td>X</td>
<td>SP, FRTp (BCD), FRBp</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>denbcd</td>
<td>DFP Encode BCD To BCD</td>
<td>X</td>
<td>S, FRT, FRB (BCD)</td>
<td>-</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>V</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>denbcdq</td>
<td>DFP Encode BCD To BCD Quad</td>
<td>X</td>
<td>S, FRTp, FRBp (BCD)</td>
<td>-</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>dxex</td>
<td>DFP Extract Biased Exponent</td>
<td>X</td>
<td>FRT (FIX), FRB</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>dxexq</td>
<td>DFP Extract Biased Exponent Quad</td>
<td>X</td>
<td>FRT (FIX), FRBp</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>dxex</td>
<td>DFP Insert Biased Exponent</td>
<td>X</td>
<td>FRT, FRA (FIX), FRB</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>dxexq</td>
<td>DFP Insert Biased Exponent Quad</td>
<td>X</td>
<td>FRTp, FRA (FIX), FRBp</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>dscli</td>
<td>DFP Shift Significand Left Immediate</td>
<td>Z22</td>
<td>FRT, FRA, SH</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>dscliq</td>
<td>DFP Shift Significand Left Immediate Quad</td>
<td>Z22</td>
<td>FRTp, FRAp, SH</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>dscri</td>
<td>DFP Shift Significand Right Immediate</td>
<td>Z22</td>
<td>FRT, FRA, SH</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>dscriq</td>
<td>DFP Shift Significand Right Immediate Quad</td>
<td>Z22</td>
<td>FRTp, FRAp, SH</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>

Explanation:

# FI and FR are set to zeros for these instructions.
- Not applicable.
1 A unique definition of the FPSCR\_PPC field is provided for the instruction.
2 These are the only instructions that may generate an SNaN and also set the FPSCR\_FPRF field. Since the BFP FPRF field does not include a code for SNaN, these instructions cause the need for redefining the FPRF field for DFP.

DCM A 6-bit immediate operand specifying the data-class mask.
DGM A 6-bit immediate operand specifying the data-group mask.
G An SNaN can be generated as the target operand.
IE An ideal exponent is defined for the instruction.
FI Setting of the FPSCR\_FI flag.
FR Setting of the FPSCR\_FR flag.
N No.
O An overflow exception may be recognized.
Rc The record bit, Rc, is provided to record FPSCR\_32:35 in CR field 1.
RE The trailing significand field is reencoded using preferred DPD encodings. The preferred DPD encoding are also used for propagated NaNs, or converted NaNs and infinities.
RMC A 2-bit immediate operand specifying the rounding-mode control.
S An one-bit immediate operand specifying if the operation is signed or unsigned.
Explanation:

- **SP**: A two-bit immediate operand: one bit specifies if the operation is signed or unsigned and, for signed operations, another bit specifies which preferred plus sign code is generated.
- **U**: An underflow exception may be recognized.
- **V**: An invalid-operation exception may be recognized.
- **Vs**: An input operand of SNaN causes an invalid-operation exception.
- **X**: An inexact exception may be recognized.
- **Y**: Yes.
- **U**: Undefined
- **Z**: A zero-divide exception may be recognized.
Chapter 6. Vector Facility

6.1 Vector Facility Overview
This chapter describes the registers and instructions that make up the Vector Facility.

6.2 Chapter Conventions

6.2.1 Description of Instruction Operation
The following notation, in addition to that described in Section 1.3.2, is used in this chapter.

- \texttt{x.bit[y]} \quad Return the contents of bit y of x.
- \texttt{x.bit[y:z]} \quad Return the contents of bits y:z of x.
- \texttt{x.nibble[y]} \quad Return the contents of the 4-bit nibble element y of x.
- \texttt{x.nibble[y:z]} \quad Return the contents of the 4-bit nibble elements y:z of x.
- \texttt{x.byte[y]} \quad Return the contents of 8-bit byte element y of x.
- \texttt{x.byte[y:z]} \quad Return the contents of 8-bit byte elements y:z of x.
- \texttt{x.hword[y]} \quad Return the contents of 16-bit halfword element y of x.
- \texttt{x.hword[y:z]} \quad Return the contents of 16-bit halfword elements y:z of x.
- \texttt{x.word[y]} \quad Return the contents of 32-bit word element y of x.
- \texttt{x.word[y:z]} \quad Return the contents of 32-bit word element y:z of x.
- \texttt{x.dword[y]} \quad Return the contents of 64-bit doubleword element y of x.
- \texttt{x.dword[y:z]} \quad Return the contents of 64-bit doubleword elements y:z of x.
- \texttt{x.qword[y]} \quad Return the contents of 128-bit quadword element y of x.
- \texttt{x ? y : z} \quad if the value of x is true, then the value of y, otherwise the value z.
- \texttt{+} \quad Addition.
- \texttt{-} \quad Subtraction.
- \texttt{\times} \quad Multiplication.
- \texttt{~} \quad One’s complement.
- \texttt{=, <, <=, >, >=} \quad Equal, less than, less than or equal, greater than, and greater than or equal comparison relations.
\( x \ll y \)
Result of shifting \( x \) left by \( y \) bits, filling vacated bits with zeros.

\[
b \leftarrow \text{LENGTH}(x) \\
\text{result} \leftarrow x \\
do ~ i = 0 \text{ to } y-1 \\
\quad \text{result} \leftarrow \text{result}.\text{bit}[1:b-1] \ || \ 0b0
\]

\( x \gg y \)
Result of shifting \( x \) right by \( y \) bits, filling vacated bits with copies of bit 0 of \( x \).

\[
b \leftarrow \text{LENGTH}(x) \\
\text{result} \leftarrow x \\
do ~ i = 0 \text{ to } y-1 \\
\quad \text{result} \leftarrow \text{result}.\text{bit}[0] \ || \ \text{result}.\text{bit}[0:b-2]
\]

\( x \lll y \)
Result of rotating \( x \) left by \( y \) bits.

\[
b \leftarrow \text{LENGTH}(x) \\
\text{result} \leftarrow x.\text{bit}[y:b-1] \ || \ x.\text{bit}[0:y-1]
\]

\( x \ggg y \)
Result of rotating \( x \) right by \( y \) bits.

\[
b \leftarrow \text{LENGTH}(x) \\
\text{result} \leftarrow x.\text{bit}[b-y+1:b-1] \ || \ x.\text{bit}[0:b-y]
\]

\text{bcd}_\text{ADD}(x,y,z)
Let \( x \) and \( y \) be 31-digit signed decimal values.
Performs a signed decimal addition of \( x \) and \( y \).

If the unbounded result is equal to zero, \eq_{\text{flag}} \) is set to 1. Otherwise, \eq_{\text{flag}} \) is set to 0.
If the unbounded result is greater than zero, \gt_{\text{flag}} \) is set to 1. Otherwise, \gt_{\text{flag}} \) is set to 0.
If the unbounded result is less than zero, \lt_{\text{flag}} \) is set to 1. Otherwise, \lt_{\text{flag}} \) is set to 0.

If the magnitude of the unbounded result is greater than \( 10^{31-1} \), \ox_{\text{flag}} \) is set to 1. Otherwise, \ox_{\text{flag}} \) is set to 0.

If the unbounded result is greater than or equal to zero, the sign code of the result is set to \( 0b1100 \) if \( z=0 \).
If the unbounded result is greater than or equal to zero, the sign code of the result is set to \( 0b1111 \) if \( z=1 \).
If the unbounded result is less than zero, the sign code of the result is set to \( 0b1101 \).

The low-order 31 digits of the unbounded result magnitude concatenated with the sign code are returned.

If either operand is an invalid encoding of a signed decimal value, the result returned is undefined and \inv_{\text{flag}} \) is set to 1 and \lt_{\text{flag}}, \gt_{\text{flag}} \) and \eq_{\text{flag}} \) are set to 0. Otherwise, \inv_{\text{flag}} \) is set to 0.
bcd_CONVERT_FROM_SI128(x, y)
Let x be a signed integer quadword.
Let y indicate the preferred sign code.

Return the signed integer value x in packed decimal format.

```plaintext
if x < 0 then do
    x ← ~x + 1
    sign ← 0x000D
end
else
    sign ← (y=0) ? 0x000C : 0x000F

result ← 0
shcnt ← 4

do while (x > 0)
    digit ← x % 10
    result ← result | (digit<<shcnt)
    x ← x ÷ 10
    shcnt ← shcnt + 4
end

return result | sign
```

bcd_INCREMENT(result)
Increments the magnitude of the packed decimal value x by 1.

bcd_SUBTRACT(x, y, z)
Let x and y be 31-digit signed decimal values.

Performs a signed decimal subtract of y from x.

If the unbounded result is equal to zero, eq_flag is set to 1. Otherwise, eq_flag is set to 0.
If the unbounded result is greater than zero, gt_flag is set to 1. Otherwise, gt_flag is set to 0.
If the unbounded result is less than zero, lt_flag is set to 1. Otherwise, lt_flag is set to 0.

If the magnitude of the unbounded result is greater than 10^{31-1}, ox_flag is set to 1. Otherwise, ox_flag is set to 0.

If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1100 if z=0.
If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1111 if z=1.
If the unbounded result is less than zero, the sign code of the result is set to 0b1101.

The low-order 31 digits of the unbounded result magnitude concatenated with the sign code are returned.

If either operand is an invalid encoding of a signed decimal value, the result returned is undefined and inv_flag is set to 1 and lt_flag, gt_flag and eq_flag are set to 0. Otherwise, inv_flag is set to 0.
bfp32_ADD(x, y)

x is a binary floating-point value represented in single-precision format.
y is a binary floating-point value represented in single-precision format.

If x is a QNaN, the result is x.
Otherwise, if x is an SNaN, the result is x converted to a QNaN.
Otherwise, if y is a QNaN, the result is y.
Otherwise, if y is an SNaN, the result is y converted to a QNaN.
Otherwise, if x and y are infinities having opposite signs, the result is the single-precision standard QNaN.
Otherwise, if x is an Infinity, the result is x.
Otherwise, if y is an Infinity, the result is y.
Otherwise, the result is the sum, x added to y, rounded to the nearest single-precision value.

Return the result represented in single-precision format.

bfp32_CONVERT_FROM_SI32(x, y)

Let x be a 32-bit signed integer value.

sign ← X.bit[0]
exp ← 32 + 127
frac.bit[0] ← x.bit[0]
frac.bit[1:32] ← x.bit[0:31]

if frac=0 return 0x0000_0000     // Zero operand
if sign=1 then frac = ~frac + 1

do while (frac.bit[0]=0)
   frac ← frac << 1
   exp ← exp - 1
end

lsb ← frac.bit[23]
gbit ← frac.bit[24]
xbit ← frac.bit[25:32]!=0
inc ← (lsb & gbit) | (gbit & xbit)
frac.bit[0:23] ← frac.bit[0:23] + inc
if carry_out=1 then exp ← exp + 1

result.bit[0] ← sign
result.bit[1:8] ← exp - y

return result
\[\textbf{bfp32\_CONVERT\_FROM\_UI32}(x, y)\]

\(x\) is a 32-bit unsigned integer value.

\[
\begin{align*}
\exp & \leftarrow 31 + 127 \\
\frac{} & \leftarrow x.\text{bit}[0:31] \\
\text{if } \frac{}=0 \text{ return } 0x0000_0000 \text{ // Zero} \\
\text{do while } \frac{}=0 \\
\frac{} & \leftarrow \frac{} << 1 \\
\exp & \leftarrow \exp - 1 \\
\text{end} \\
\text{lsb} & \leftarrow \frac{}.\text{bit}[23] \\
\text{gbit} & \leftarrow \frac{}.\text{bit}[24] \\
x\text{bit} & \leftarrow \frac{}.\text{bit}[25:31]!=0 \\
\text{inc} & \leftarrow (\text{lsb} \& \text{gbit}) | (\text{gbit} \& x\text{bit}) \\
\frac{}.\text{bit}[0:23] & \leftarrow \frac{}.\text{bit}[0:23] + \text{inc} \\
\text{if carry\_out}=1 \text{ then } \exp & \leftarrow \exp + 1 \\
\text{result}.\text{bit}[0] & \leftarrow 0b0 \\
\text{result}.\text{bit}[1:8] & \leftarrow \exp - y \\
\text{result}.\text{bit}[9:31] & \leftarrow \frac{}.\text{bit}[1:23] \\
\text{return result}
\end{align*}
\]

\[\textbf{bfp32\_LOG\_BASE2\_ESTIMATE}(x)\]

\(x\) is a floating-point value represented in single-precision format.

Returns a floating-point estimate of the base 2 logarithm of \(x\), represented in single-precision format.

\[\textbf{bfp32\_MAXIMUM}(x, y)\]

\(x\) is a floating-point value represented in single-precision format.
\(y\) is a floating-point value represented in single-precision format.

Returns the largest value of \(x\) and \(y\), represented in single-precision format.
- The maximum of \(+0.0\) and \(-0.0\) is \(+0.0\).
- The maximum of any value and a NaN is a QNaN.

\[\textbf{bfp32\_MINIMUM}(x, y)\]

\(x\) is a floating-point value represented in single-precision format.
\(y\) is a floating-point value represented in single-precision format.

Returns the smallest value of \(x\) and \(y\), represented in single-precision format.
- The minimum of \(+0.0\) and \(-0.0\) is \(-0.0\).
- The minimum of any value and a NaN is a QNaN.
**bfp32_MULTIPLY_ADD**(x, z, y)

- x is a binary floating-point value represented in single-precision format.
- y is a binary floating-point value represented in single-precision format.
- z is a binary floating-point value represented in single-precision format.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, the result is x converted to a QNaN.
Otherwise, if y is a QNaN, the result is y.
Otherwise, if y is an SNaN, the result is y converted to a QNaN.
Otherwise, if z is a QNaN, the result is z.
Otherwise, if z is an SNaN, the result is z converted to a QNaN.
Otherwise, if x is an Infinity and z is a Zero, the result is the single-precision standard QNaN.
Otherwise, if x is a Zero and z is an Infinity, the result is the single-precision standard QNaN.
Otherwise, if the product, x multiplied by z, and y are Infinities having opposite signs, the result is the single-precision standard QNaN.
Otherwise, the result is the sum of the product, x multiplied by z, added to y, rounded to the nearest single-precision value.

Return the result represented in single-precision format.

**bfp32_NEGATIVE_MULTIPLY_SUBTRACT**(x, z, y)

- x is a binary floating-point value represented in single-precision format.
- y is a binary floating-point value represented in single-precision format.
- z is a binary floating-point value represented in single-precision format.

If x is a QNaN, the result is x.
Otherwise, if x is an SNaN, the result is x converted to a QNaN.
Otherwise, if y is a QNaN, the result is y.
Otherwise, if y is an SNaN, the result is y converted to a QNaN.
Otherwise, if z is a QNaN, the result is z.
Otherwise, if z is an SNaN, the result is z converted to a QNaN.
Otherwise, if x is an Infinity and z is a Zero, the result is the single-precision standard QNaN.
Otherwise, if x is a Zero and z is an Infinity, the result is the single-precision standard QNaN.
Otherwise, if the product, x multiplied by z, and y are Infinities having the same signs, the result is the single-precision standard QNaN.
Otherwise, the result is the difference of the product, x multiplied by z, subtracted by y, then rounded to the nearest single-precision value, and then negated.

Return the result represented in single-precision format.

**bfp32_POWER2_ESTIMATE**(x)

- x is a floating-point value represented in single-precision format.

Returns a floating-point estimate of 2 raised to the power of x, represented in single-precision format.

**bfp32_RECIPROCAL_ESTIMATE**(x)

- x is a floating-point value represented in single-precision format.

Returns a floating-point estimate of the reciprocal of x, represented in single-precision format.

**bfp32_RECIPROCAL_SQRT_ESTIMATE**(x)

- x is a floating-point value represented in single-precision format.

Returns a floating-point estimate of the reciprocal of the square root of x, represented in single-precision format.
bfp32_ROUND_TO_INTEGER_CEIL(x)

x is a floating-point value represented in single-precision format.

Returns the smallest floating-point integer that is greater than or equal to x, represented in single-precision format.

bfp32_ROUND_TO_INTEGER_FLOOR(x)

x is a floating-point value represented in single-precision format.

Returns the largest floating-point integer that is less than or equal to x, represented in single-precision format.

bfp32_ROUND_TO_INTEGER_NEAR(x)

x is a floating-point value represented in single-precision format.

Returns the floating-point integer that is nearest to x (in case of a tie, the even single-precision floating-point integer is used), represented in single-precision format.

bfp32_ROUND_TO_INTEGER_TRUNC(x)

x is a floating-point value represented in single-precision format.

Returns the largest floating-point integer that is less than or equal to x if x>0, or the smallest floating-point integer that is greater than or equal to x if x>0, or represented in single-precision format.

bfp32_SUBTRACT(x,y)

x is a binary floating-point value represented in single-precision format.
y is a binary floating-point value represented in single-precision format.

If x is a QNaN, the result is x.
Otherwise, if x is an SNaN, the result is x converted to a QNaN.
Otherwise, if y is a QNaN, the result is y.
Otherwise, if y is an SNaN, the result is y converted to a QNaN.
Otherwise, if x and y are infinities having the same signs, the result is the single-precision standard QNaN.
Otherwise, if x is an infinity, the result is x.
Otherwise, if y is an infinity, the result is y.
Otherwise, the result is the difference, x subtracted by y, rounded to the nearest single-precision value.

Return the result represented in single-precision format.

bool_COMPARE_GE_BFP32(x,y)

x is a floating-point value represented in the single-precision format.
y is a floating-point value represented in the single-precision format.

Returns the value 1 if x is greater than or equal to y. Otherwise, returns the value 0.

bool_COMPARE_GT_BFP32(x,y)

x is a floating-point value represented in the single-precision format.
y is a floating-point value represented in the single-precision format.

Returns the value 1 if x is greater than y. Otherwise, returns the value 0.
bool_COMPARE_EQ_BFP32(x, y)
  x is a floating-point value represented in the single-precision format.
  y is a floating-point value represented in the single-precision format.

  Returns the value 1 if x is equal to y. Otherwise, returns the value 0.

bool_COMPARE_LE_BFP32(x, y)
  x is a floating-point value represented in the single-precision format.
  y is a floating-point value represented in the single-precision format.

  Returns the value 1 if x is less than or equal to y. Otherwise, returns the value 0.

CHOP8(x)
  Returns rightmost 8 bits of x padded on the left with zeros if necessary.

CHOP16(x)
  Returns rightmost 16 bits of x padded on the left with zeros if necessary.

CHOP32(x)
  Returns rightmost 32 bits of x padded on the left with zeros if necessary.

CHOP64(x)
  Returns rightmost 64 bits of x padded on the left with zeros if necessary.

CHOP128(x)
  Returns rightmost 128 bits of x padded on the left with zeros if necessary.

Clamp(x, y, z)
  x is interpreted as a signed integer. If the value of x is less than y, then the value y is returned, else if the value of x is greater than z, the value z is returned, else the value x is returned.

  if x < y then
    result ← y
  else if x > z then
    result ← z
  else
    result ← x

EXTS(x)
  Result of extending x on the left with copies of bit 0 of x to form a signed integer value having unbounded range.

EXTSB(x)
  Result of extending x on the left with copies of bit 0 of x to form an 8-bit signed integer value.

EXTS16(x)
  Result of extending x on the left with copies of bit 0 of x to form a 16-bit signed integer value.

EXTS32(x)
  Result of extending x on the left with copies of bit 0 of x to form a 32-bit signed integer value.

EXTS64(x)
  Result of extending x on the left with copies of bit 0 of x to form a 64-bit signed integer value.

EXTS128(x)
  Result of extending x on the left with copies of bit 0 of x to form a 128-bit signed integer value.

EXTZ(x)
  Result of extending x on the left with 0s to form a positive signed integer value having unbounded range.
EXTZB(x)
Result of extending x on the left with 0s to form an 8-bit unsigned integer value.

EXTZ16(x)
Result of extending x on the left with 0s to form a 16-bit unsigned integer value.

EXTZ32(x)
Result of extending x on the left with 0s to form a 32-bit unsigned integer value.

EXTZ64(x)
Result of extending x on the left with 0s to form a 64-bit unsigned integer value.

EXTZ128(x)
Result of extending x on the left with 0s to form a 128-bit unsigned integer value.

InvMixColumns(x)
\[
\text{InvMixColumns}(x) = \begin{cases}
\text{result.word}[c].\text{byte}[0] &= 0x0E \times x.\text{word}[c].\text{byte}[0] \oplus 0x0B \times x.\text{word}[c].\text{byte}[1] \\
\text{result.word}[c].\text{byte}[1] &= 0x09 \times x.\text{word}[c].\text{byte}[0] \oplus 0x0E \times x.\text{word}[c].\text{byte}[1] \\
\text{result.word}[c].\text{byte}[2] &= 0x0D \times x.\text{word}[c].\text{byte}[0] \oplus 0x09 \times x.\text{word}[c].\text{byte}[1] \\
\text{result.word}[c].\text{byte}[3] &= 0x0B \times x.\text{word}[c].\text{byte}[0] \oplus 0x0D \times x.\text{word}[c].\text{byte}[1]
\end{cases}
\]
end
return result;

where \(\times\) is a GF(\(2^8\)) multiply, a binary polynomial multiplication reduced by modulo \(0x11B\).

The GF(\(2^8\)) multiply of \(0x09 \times x\) can be expressed in minimized terms as the following.
\[
\text{product.bit}[0] = x.\text{bit}[0] \oplus x.\text{bit}[3] \\
\text{product.bit}[1] = x.\text{bit}[1] \oplus x.\text{bit}[4] \oplus x.\text{bit}[0] \\
\text{product.bit}[2] = x.\text{bit}[2] \oplus x.\text{bit}[5] \oplus x.\text{bit}[0] \oplus x.\text{bit}[1] \\
\text{product.bit}[3] = x.\text{bit}[3] \oplus x.\text{bit}[6] \oplus x.\text{bit}[1] \oplus x.\text{bit}[2] \\
\text{product.bit}[4] = x.\text{bit}[4] \oplus x.\text{bit}[7] \oplus x.\text{bit}[0] \oplus x.\text{bit}[2] \\
\text{product.bit}[5] = x.\text{bit}[5] \oplus x.\text{bit}[0] \oplus x.\text{bit}[1] \\
\text{product.bit}[6] = x.\text{bit}[6] \oplus x.\text{bit}[1] \oplus x.\text{bit}[2] \\
\text{product.bit}[7] = x.\text{bit}[7] \oplus x.\text{bit}[0] \\
\]

The GF(\(2^8\)) multiply of \(0x0B \times x\) can be expressed in minimized terms as the following.
\[
\text{product.bit}[0] = x.\text{bit}[0] \oplus x.\text{bit}[1] \oplus x.\text{bit}[3] \\
\text{product.bit}[1] = x.\text{bit}[1] \oplus x.\text{bit}[2] \oplus x.\text{bit}[4] \oplus x.\text{bit}[0] \\
\text{product.bit}[2] = x.\text{bit}[2] \oplus x.\text{bit}[3] \oplus x.\text{bit}[5] \oplus x.\text{bit}[0] \oplus x.\text{bit}[1] \\
\text{product.bit}[3] = x.\text{bit}[3] \oplus x.\text{bit}[6] \oplus x.\text{bit}[1] \oplus x.\text{bit}[2] \\
\text{product.bit}[4] = x.\text{bit}[4] \oplus x.\text{bit}[7] \oplus x.\text{bit}[0] \oplus x.\text{bit}[2] \\
\text{product.bit}[5] = x.\text{bit}[5] \oplus x.\text{bit}[6] \oplus x.\text{bit}[0] \oplus x.\text{bit}[1] \\
\text{product.bit}[6] = x.\text{bit}[6] \oplus x.\text{bit}[7] \oplus x.\text{bit}[0] \oplus x.\text{bit}[1] \oplus x.\text{bit}[2] \\
\text{product.bit}[7] = x.\text{bit}[7] \oplus x.\text{bit}[0] \\
\]

The GF(\(2^8\)) multiply of \(0x0D \times x\) can be expressed in minimized terms as the following.
\[
\text{product.bit}[0] = x.\text{bit}[0] \oplus x.\text{bit}[2] \oplus x.\text{bit}[3] \\
\text{product.bit}[1] = x.\text{bit}[1] \oplus x.\text{bit}[3] \oplus x.\text{bit}[4] \oplus x.\text{bit}[6] \\
\text{product.bit}[2] = x.\text{bit}[2] \oplus x.\text{bit}[4] \oplus x.\text{bit}[5] \oplus x.\bit[1] \\
\text{product.bit}[3] = x.\text{bit}[3] \oplus x.\text{bit}[6] \oplus x.\text{bit}[7] \oplus x.\text{bit}[0] \oplus x.\text{bit}[1] \oplus x.\text{bit}[2] \\
\text{product.bit}[4] = x.\text{bit}[4] \oplus x.\text{bit}[7] \oplus x.\text{bit}[0] \oplus x.\text{bit}[2] \\
\text{product.bit}[5] = x.\text{bit}[5] \oplus x.\text{bit}[7] \oplus x.\text{bit}[1] \\
\text{product.bit}[6] = x.\text{bit}[6] \oplus x.\text{bit}[0] \oplus x.\text{bit}[2] \\
\text{product.bit}[7] = x.\text{bit}[7] \oplus x.\text{bit}[0] \oplus x.\text{bit}[1] \oplus x.\text{bit}[2]
The $GF(2^8)$ multiply of $0x0E\times x$ can be expressed in minimized terms as the following.

\[
\begin{align*}
\text{product.bit[0]} &= x.\text{bit[1]} \land x.\text{bit[3]} \land x.\text{bit[2]} \\
\text{product.bit[1]} &= x.\text{bit[3]} \land x.\text{bit[4]} \land x.\text{bit[0]} \\
\text{product.bit[2]} &= x.\text{bit[4]} \land x.\text{bit[5]} \land x.\text{bit[2]} \\
\text{product.bit[3]} &= x.\text{bit[5]} \land x.\text{bit[6]} \land x.\text{bit[1]} \\
\text{product.bit[4]} &= x.\text{bit[6]} \land x.\text{bit[7]} \land x.\text{bit[1]} \\
\text{product.bit[5]} &= x.\text{bit[7]} \land x.\text{bit[5]} \\
\text{product.bit[6]} &= x.\text{bit[0]} \land x.\text{bit[1]} \land x.\text{bit[2]}
\end{align*}
\]

```python
InvShiftRows(x):
    result.word[0].byte[0] = x.word[0].byte[0]
    result.word[1].byte[0] = x.word[1].byte[0]
    result.word[2].byte[0] = x.word[2].byte[0]
    result.word[3].byte[0] = x.word[3].byte[0]
    result.word[0].byte[1] = x.word[3].byte[1]
    result.word[1].byte[1] = x.word[0].byte[1]
    result.word[0].byte[3] = x.word[1].byte[3]
    return(result)
```

```python
InvSubBytes(x):
    InvSBOX.byte[256] = { .... }  
    do i = 0 to 15
        result.byte[i] = InvSBOX.byte[x.byte[i]]
    end
    return(result)
```

```python
LENGTH(x):
    Length of x, in bits. If x is the word "element", LENGTH(x) is the length, in bits, of the element implied by the instruction mnemonic.
```
MASK128(x, y)
Let x and y be integer values from 0 to 127.
Generate a 128-bit mask that consists of 1-bits from a start bit, x, through and including a stop bit, y, and 0-bits elsewhere.

\[
\text{if } x \leq y \text{ then } \\
\text{mask} = \text{all 0s} \\
\text{mask}.bit[x:y] = \text{all 1s} \\
\text{else} \\
\text{mask} = \text{all 1s} \\
\text{mask}.bit[y+1:x-1] = \text{all 0s} \\
\text{return mask }
\]

MixColumns(x)
do c = 0 to 3
result.word[c].byte[0] = 0x02•x.word[c].byte[0] ^ 0x03•x.word[c].byte[1] ^ x.word[c].byte[2] ^ x.word[c].byte[3] 
result.word[c].byte[1] = x.word[c].byte[0] ^ 0x02•x.word[c].byte[1] ^ 0x03•x.word[c].byte[2] ^ x.word[c].byte[3] 
result.word[c].byte[2] = x.word[c].byte[0] ^ x.word[c].byte[1] ^ 0x02•x.word[c].byte[2] ^ 0x03•x.word[c].byte[3] 
result.word[c].byte[3] = 0x03•x.word[c].byte[0] ^ x.word[c].byte[1] ^ x.word[c].byte[2] ^ 0x02•x.word[c].byte[3] 
end 
return(result)

The GF(2^8) multiply of 0x02•x can be expressed in minimized terms as the following.

\[
\text{product}.bit[0] = x.bit[1] \\
\text{product}.bit[7] = x.bit[0]
\]

The GF(2^8) multiply of 0x03•x can be expressed in minimized terms as the following.

\[
\text{product}.bit[0] = x.bit[0] ^ x.bit[1] \\
\text{product}.bit[7] = x.bit[0]
\]

qword_bit_splat(x)
x is a 1-bit value.
Return the concatenation of 128 copies of x.

ROTL128(x, y)
Let x be a 128-bit integer value.
Let y be an integer value.
Return x rotated left by y bits.
**ShiftRows(x)**

```plaintext
result.word[0].byte[0] = x.word[0].byte[0]
result.word[1].byte[0] = x.word[1].byte[0]
result.word[2].byte[0] = x.word[2].byte[0]
result.word[3].byte[0] = x.word[3].byte[0]

result.word[0].byte[1] = x.word[1].byte[1]
result.word[3].byte[1] = x.word[0].byte[1]


result.word[1].byte[3] = x.word[0].byte[3]
```

**si8_CLAMP(x)**

Let \( x \) be a signed integer value.

Return the value \( x \) in 8-bit signed integer format.

- If the value of the element is greater than \( 2^{7} - 1 \) the result saturates to \( 2^{7} - 1 \) and \( \text{sat\_flag} \) is set to 1.
- If the value of the element is less than \( -2^{7} \) the result saturates to \( -2^{7} \) and \( \text{sat\_flag} \) is set to 1.

**si16_CLAMP(x)**

Let \( x \) be a signed integer value.

Return the value \( x \) in 16-bit signed integer format.

- If the value of the element is greater than \( 2^{15} - 1 \) the result saturates to \( 2^{15} - 1 \) and \( \text{SAT} \) is set to 1.
- If the value of the element is less than \( -2^{15} \) the result saturates to \( -2^{15} \) and \( \text{SAT} \) is set to 1.

**si32_CLAMP(x)**

Let \( x \) be a signed integer value.

Return the value \( x \) in 32-bit signed integer format.

- If the value of the element is greater than \( 2^{31} - 1 \) the result saturates to \( 2^{31} - 1 \) and \( \text{SAT} \) is set to 1.
- If the value of the element is less than \( -2^{31} \) the result saturates to \( -2^{31} \) and \( \text{SAT} \) is set to 1.
si32_CONVERT_FROM_BFP32(x, y)

Let \( x \) be a single-precision floating-point value.
Let \( y \) be an unsigned integer value.

\[
\begin{align*}
\text{sign} & \leftarrow x._{bit}[0] \\
\text{exp} & \leftarrow x._{bit}[1:8] \\
\text{frac}._{bit}[0:22] & \leftarrow x._{bit}[9:31] \\
\text{frac}._{bit}[23:30] & \leftarrow 0b0000_0000 \\
\text{if exp}=255 & \& \text{frac}!\neq0 \text{ then return } 0x0000_0000 & \quad \text{// NaN operand} \\
\text{if exp}=255 & \& \text{frac}=0 \text{ then do} & \quad \text{// infinity operand} \\
& \quad \text{VSCR.SAT} \leftarrow 1 \\
& \quad \text{return } (\text{sign}=1) ? 0x8000_0000 : 0x7FFF_FFFF \\
\end{align*}
\]

\[
\begin{align*}
\text{if (exp}+y-127) > 30 \text{ then do} & \quad \text{// large operand} \\
& \quad \text{VSCR.SAT} \leftarrow 1 \\
& \quad \text{return } (\text{sign}=1) ? 0x8000_0000 : 0x7FFF_FFFF \\
\end{align*}
\]

\[
\begin{align*}
\text{if (exp}+y-127) < 0 \text{ then return } 0x0000_0000 & \quad \text{// -1.0 < value < 1.0 (value rounds to 0)} \\
\text{significand}._{bit}[0:31] & \leftarrow 0x0000_0000 \\
\text{significand}._{bit}[32] & \leftarrow 0x1 \\
\text{significand}._{bit}[33:63] & \leftarrow \text{frac} \\
\text{do } i = 1 \text{ to } 31-(\text{exp}+y-127) & \quad \text{significand} \leftarrow \text{significand} \gg 1 \\
\text{end} \\
\text{return } (\text{sign}=0) ? \text{CHOP32(significand)} : \text{CHOP32(-significand} + 1)
\end{align*}
\]

si128_CONVERT_FROM_BCD(x)

Let \( x \) be a packed decimal value.

Return the value \( x \) in 128-bit signed integer format.

\[
\begin{align*}
\text{result} & \leftarrow 0 \\
\text{scale} & \leftarrow 1 \\
\text{sign} & \leftarrow x._{bit}[124:127] \\
\text{x} & \leftarrow 0b0000 || x._{nibble}[0:30] \\
\text{do while } x > 0 & \quad \text{digit} \leftarrow x \& 0x000F \\
& \quad \text{result} \leftarrow \text{result} + (\text{digit} \times \text{scale}) \\
& \quad x \leftarrow 0b0000 || x._{nibble}[0:30] \\
& \quad \text{scale} \leftarrow \text{scale} \times 10 \\
\end{align*}
\]

\[
\begin{align*}
\text{if sign}=0x0008 & \| \text{sign}=0x000D \text{ then} \\
& \quad \text{return } \neg \text{result} + 1 \\
\end{align*}
\]
Version 3.1
SubBytes(x)
SBOX.byte[0:255] = { 0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE,0xD7,0xAB,0x76,
0xCA,0x82,0xC9,0x7D,0xFA,0x59,0x47,0xF0,0xAD,0xD4,0xA2,0xAF,0x9C,0xA4,0x72,0xC0,
0xB7,0xFD,0x93,0x26,0x36,0x3F,0xF7,0xCC,0x34,0xA5,0xE5,0xF1,0x71,0xD8,0x31,0x15,
0x04,0xC7,0x23,0xC3,0x18,0x96,0x05,0x9A,0x07,0x12,0x80,0xE2,0xEB,0x27,0xB2,0x75,
0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6,0xB3,0x29,0xE3,0x2F,0x84,
0x53,0xD1,0x00,0xED,0x20,0xFC,0xB1,0x5B,0x6A,0xCB,0xBE,0x39,0x4A,0x4C,0x58,0xCF,
0xD0,0xEF,0xAA,0xFB,0x43,0x4D,0x33,0x85,0x45,0xF9,0x02,0x7F,0x50,0x3C,0x9F,0xA8,
0x51,0xA3,0x40,0x8F,0x92,0x9D,0x38,0xF5,0xBC,0xB6,0xDA,0x21,0x10,0xFF,0xF3,0xD2,
0xCD,0x0C,0x13,0xEC,0x5F,0x97,0x44,0x17,0xC4,0xA7,0x7E,0x3D,0x64,0x5D,0x19,0x73,
0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,0x5E,0x0B,0xDB,
0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C,0xC2,0xD3,0xAC,0x62,0x91,0x95,0xE4,0x79,
0xE7,0xC8,0x37,0x6D,0x8D,0xD5,0x4E,0xA9,0x6C,0x56,0xF4,0xEA,0x65,0x7A,0xAE,0x08,
0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0x8B,0x8A,
0x70,0x3E,0xB5,0x66,0x48,0x03,0xF6,0x0E,0x61,0x35,0x57,0xB9,0x86,0xC1,0x1D,0x9E,
0xE1,0xF8,0x98,0x11,0x69,0xD9,0x8E,0x94,0x9B,0x1E,0x87,0xE9,0xCE,0x55,0x28,0xDF,
0x8C,0xA1,0x89,0x0D,0xBF,0xE6,0x42,0x68,0x41,0x99,0x2D,0x0F,0xB0,0x54,0xBB,0x16 }
do i = 0 to 15
result.byte[i] = SBOX.byte[x.byte[i]]
end
return(result)

ui8_CLAMP(x)
Let x be a signed integer value.
Return the value x in 8-bit unsigned integer format.
– If the value of the element is greater than 28-1 the result saturates to 28-1 and SAT is set to 1.
– If the value of the element is less than 0 the result saturates to 0 and SAT is set to 1.
ui16_CLAMP(x)
Let x be a signed integer value.
Return the value x in 16-bit unsigned integer format.
– If the value of the element is greater than 216-1 the result saturates to 216-1 and SAT is set to 1.
– If the value of the element is less than 0 the result saturates to 0 and SAT is set to 1.
ui32_CLAMP(x)
Let x be a signed integer value.
Return the value x in 32-bit unsigned integer format.
– If the value of the element is greater than 232-1 the result saturates to 232-1 and SAT is set to 1.
– If the value of the element is less than 0 the result saturates to 0 and SAT is set to 1.

256

Power ISA™ I


ui32_CONVERT_FROM_BFP32(x, y)
Let x be a single-precision floating-point value.
Let y be an unsigned integer value.

sign ← x.bit[0]
exp ← x.bit[1:8]
frac.bit[0:22] ← x.bit[9:31]
frac.bit[23:30] ← 0b0000_0000
if exp=255 & frac!=0 then return 0x0000_0000     // NaN operand
if exp=255 & frac=0 then do                      // infinity operand
   VSCR.SAT ← 1
   return (sign=1) ? 0x0000_0000 : 0xFFFF_FFFF
end
if (exp+Y-127)>31 then do                        // large operand
   VSCR.SAT ← 1
   return (sign=1) ? 0x0000_0000 : 0xFFFF_FFFF
end
if (exp+Y-127) < 0 then return 0x0000_0000       // -1.0 < value < 1.0
   // value rounds to 0
if sign=1 then do                                // negative operand
   VSCR.SAT ← 1
   return 0x0000_0000
end
significand.bit[0:31] ← 0x0000_0000
significand.bit[32] ← 0b1
significand.bit[33:63] ← frac
do i = 1 to 31-(exp+Y-127)
   significand = significand >> 1
end
return CHOP32(significand)
6.3 Vector Facility Registers

6.3.1 Vector-Scalar Registers

The Vector instructions described in Chapter 6 are defined to operate on the higher-numbered 32 Vector-Scalar Registers (VSRs 32-63), formerly known as Vector Registers (VRs 0-31). See Figure 98. All computations and other data manipulation are performed on data residing in VSRs 32-63, and results are placed into one of VSRs 32-63.

Depending on the instruction, the contents of a VSR are interpreted as a sequence of equal-length elements (bytes, halfwords, or words) or as a quadword. Each of the elements is aligned within the VSR, as shown in Figure 97. Many instructions perform a given operation in parallel on all elements in a VSR. Depending on the instruction, a byte, halfword, or word element can be interpreted as a signed-integer, an unsigned-integer, or a logical value; a word element can also be interpreted as a single-precision floating-point value. In the instruction descriptions, phrases like “signed-integer word element” are used as shorthand for “word element, interpreted as a signed-integer”.

Load and Store instructions are provided that transfer a byte, halfword, word, or quadword between storage and a VSR.

6.3.2 Vector Status and Control Register

The Vector Status and Control Register (VSCR) is a special 32-bit register (not an SPR) that is read and written in a manner similar to the FPSCR in the Power ISA scalar floating-point unit. Special instructions (mfvscr and mtvscr) are provided to move the VSCR from and to a VSR. When moved to or from a VSR, the 32-bit VSCR is right justified in the 128-bit VSR. When moved to a VSR, bits 0:95 of the VSR are cleared (set to 0).

The bit definitions for the VSCR are as follows.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>96:110</td>
<td>Reserved</td>
</tr>
<tr>
<td>111</td>
<td><strong>Vector Non-Java Mode (NJ)</strong></td>
</tr>
<tr>
<td>0</td>
<td>Denormalized values are handled by Vector Floating-Point instructions.</td>
</tr>
<tr>
<td>1</td>
<td>If an element in a source VSR contains a denormalized value, the value 0 is used instead. If an instruction causes an Underflow Exception, the corresponding element in the target VSR is set to 0. In both cases the 0 has the same sign as the denormalized or underflowing value.</td>
</tr>
</tbody>
</table>

VSR[32] (formerly VR[0])
VSR[33] (formerly VR[1])
...
VSR[62] (formerly VR[30])
VSR[63] (formerly VR[31])

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 127

VSCR
Vector Saturation ($\text{SAT}$)

Every vector instruction having “Saturate” in its name implicitly sets this bit to 1 if any result of that instruction “saturates”; see Section 6.8. $\text{mtvscr}$ can alter this bit explicitly. This bit is sticky; that is, once set to 1 it remains set to 1 until it is set to 0 by an $\text{mtvscr}$ instruction.

After the $\text{mtvscr}$ instruction executes, the result in the target VSR will be architecturally precise. That is, it will reflect all updates to the $\text{SAT}$ bit that could have been made by vector instructions logically preceding it in the program flow, and further, it will not reflect any $\text{SAT}$ updates that may be made to it by vector instructions logically following it in the program flow. To implement this, processors may choose to make the $\text{mtvscr}$ instruction execution serializing within the vector unit, meaning that it will stall vector instruction execution until all preceding vector instructions are complete and have updated the architectural machine state. This is permitted in order to simplify implementation of the sticky status bit ($\text{SAT}$) which would otherwise be difficult to implement in an out-of-order execution machine. The implication of this is that reading the VSCR can be much slower than typical Vector instructions, and therefore care must be taken in reading it, as advised in Section 6.5.1, to avoid performance problems.

The $\text{mtvscr}$ is context synchronizing. This implies that all Vector instructions logically preceding an $\text{mtvscr}$ in the program flow will execute in the architectural context ($\text{NJ}$ mode) that existed prior to completion of the $\text{mtvscr}$, and that all instructions logically following the $\text{mtvscr}$ will execute in the new context ($\text{NJ}$ mode) established by the $\text{mtvscr}$.

## 6.3.3 VR Save Register

The VR Save Register ($\text{VRSAVE}$) is a 32-bit register in the fixed-point processor provided for application and operating system use; see Section 3.2.3.

### Programming Note

$\text{VRSAVE}$ can be used to indicate which VSRs are currently being used by a program. If this is done, the operating system could save only those VSRs when an “interrupt” occurs (see Book III), and could restore only those VSRs when resuming the interrupted program.

If this approach is taken it must be applied rigorously; if a program fails to indicate that a given VSR is in use, software errors may occur that will be difficult to detect and correct because they are timing-dependent.

Some operating systems save and restore $\text{VRSAVE}$ only for programs that also use other VSRs.
6.4 Vector Storage Access Operations

The Vector Storage Access instructions provide the means by which data can be copied from storage to a VSR or from a VSR to storage. Instructions are provided that access byte, halfword, word, and quadword storage operands. These instructions differ from the fixed-point and floating-point Storage Access instructions in that vector storage operands are assumed to be aligned, and vector storage accesses are performed as if the appropriate number of low-order bits of the specified effective address (EA) were zero. For example, the low-order bit of EA is ignored for halfword Vector Storage Access instructions, and the low-order four bits of EA are ignored for quadword Vector Storage Access instructions. The effect is to load or store the storage operand of the specified length that contains the byte addressed by EA.

If a storage operand is unaligned, additional instructions must be used to ensure that the operand is correctly placed in a VSR or in storage. Instructions are provided that shift and merge the contents of two VSRs, such that an unaligned quadword storage operand can be copied between storage and the VSRs in a relatively efficient manner.

As shown in Figure 97, the elements in VSRs are numbered; the high-order (or most significant) byte element is numbered 0 and the low-order (or least significant) byte element is numbered 15. The numbering affects the values that must be placed into the permute control vector for the Vector Permute instruction in order for that instruction to achieve the desired effects, as illustrated by the examples in the following subsections.

A vector quadword Load instruction for which the effective address (EA) is quadword-aligned places the byte in storage addressed by EA into byte element 0 of the target VSR, the byte in storage addressed by EA+1 into byte element 1 of the target VSR, etc. Similarly, a vector quadword Store instruction for which the EA is quadword-aligned places the contents of byte element 0 of the source VSR into the byte in storage addressed by EA, the contents of byte element 1 of the source VSR into the byte in storage addressed by EA+1, etc.

Figure 100 shows an aligned quadword in storage. Figure 101 shows the result of loading that quadword into a VSR or, equivalently, shows the contents that must be in a VSR if storing that VSR is to produce the storage contents shown in Figure 100.

When an aligned byte, halfword, or word storage operand is loaded into a VSR, the element (byte, halfword, or word respectively) that receives the data is the element that would have received the data had the entire aligned quadword containing the storage operand addressed by EA been loaded. Similarly, when a byte, halfword, or word element in a VSR is stored into an aligned storage operand (byte, halfword, or word respectively), the element selected to be stored is the element that would have been stored into the storage operand addressed by EA had the entire VSR been stored to the aligned quadword containing the storage operand addressed by EA. (Byte storage operands are always aligned.)

For aligned byte, halfword, and word storage operands, if the corresponding element number is known when the program is written, the appropriate Vector Splat and Vector Permute instructions can be used to copy or replicate the data contained in the storage operand after loading the operand into a VSR. An example of this is given in the Programming Note for Vector Splat; see page 292. Another example is to replicate the element across an entire VSR before storing it into an arbitrary aligned storage operand of the same length; the replication ensures that the correct data are stored regardless of the offset of the storage operand in its aligned quadword in storage.

---

**Figure 100.** Aligned quadword storage operand

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
<th>08</th>
<th>09</th>
<th>0A</th>
<th>0B</th>
<th>0C</th>
<th>0D</th>
<th>0E</th>
<th>0F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

**Figure 101.** VSR contents for aligned quadword Load or Store

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
<th>08</th>
<th>09</th>
<th>0A</th>
<th>0B</th>
<th>0C</th>
<th>0D</th>
<th>0E</th>
<th>0F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>
### Figure 102. Unaligned quadword storage operand

<table>
<thead>
<tr>
<th>Vhi</th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vlo</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
</tr>
<tr>
<td>Vt, Vs</td>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
</tr>
</tbody>
</table>

### Figure 103. VSR contents

<table>
<thead>
<tr>
<th>Vhi</th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vlo</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
</tr>
<tr>
<td>Vt, Vs</td>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
</tr>
</tbody>
</table>

0 15
6.4.1 Accessing Unaligned Storage Operands

Figure 102 shows an unaligned quadword storage operand that spans two aligned quadwords. In the remainder of this section, the aligned quadword that contains the most significant bytes of the unaligned quadword is called the most significant quadword (MSQ) and the aligned quadword that contains the least significant bytes of the unaligned quadword is called the least significant quadword (LSQ). Because the Vector Storage Access instructions ignore the low-order bits of the effective address, the unaligned quadword cannot be transferred between storage and a VSR using a single instruction. The remainder of this section gives examples of accessing unaligned quadword storage operands. Similar sequences can be used to access unaligned halfword and word storage operands.

**Programming Note**

The sequence of instructions given below is one approach that can be used to load the unaligned quadword shown in Figure 102 into a VSR. In Figure 103 Vhi and Vlo are the VSRs that will receive the most significant quadword and least significant quadword respectively. VSR[VRT+32] is the target VSR.

After the two quadwords have been loaded into Vhi and Vlo, using Load Vector Indexed instructions, the alignment is performed by shifting the 32-byte quantity Vhi || Vlo left by an amount determined by the address of the first byte of the desired data. The shifting is done using a Vector Permute instruction for which the permute control vector is generated by a Load Vector for Shift Left instruction. The Load Vector for Shift Left instruction uses the same address specification as the Load Vector Indexed instruction that loads the Vhi register; this is the address of the desired unaligned quadword.

The following sequence of instructions copies the unaligned quadword storage operand into register Vt.

```assembly
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx    Vhi,0,Rb       # load MSQ
lvsr   Vp,0,Rb        # set permute control vector
addi   Rb,Rb,16       # address of LSQ
lvx    Vlo,0,Rb       # load LSQ
vperm  V1s,Vhi,Vlo,Vp # align the data
```

The procedure for storing an unaligned quadword is essentially the reverse of the procedure for loading one. However, a read-modify-write sequence is required that inserts the source quadword into two aligned quadwords in storage. The quadword to be stored is assumed to be in Vs; see Figure 103. The contents of Vs are shifted right and split into two parts, each of which is merged (using a Vector Select instruction) with the current contents of the two aligned quadwords (MSQ and LSQ) that will contain the most significant bytes and least significant bytes, respectively, of the unaligned quadword. The resulting two quadwords are stored using Store Vector Indexed instructions. A Load Vector for Shift Right instruction is used to generate the permute control vector that is used for the shifting. A single register is used for the “shifted” contents; this is possible because the “shifting” is done by means of a right rotation. The rotation is accomplished by specifying Vs for both components of the Vector Permute instruction. In addition, the same permute control vector is used on a sequence of 1s and 0s to generate the mask used by the Vector Select instructions that do the merging.

The following sequence of instructions copies the contents of Vs into an unaligned quadword in storage.

```assembly
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx    Vhi,0,Rb       # load current MSQ
lvsr   Vp,0,Rb        # set permute control vector
addi   Rb,Rb,16       # address of LSQ
lvx    Vlo,0,Rb       # load current LSQ
vspltsb Vls,-1        # generate the select mask bits
vspltsb Vls,0
vperm  Vmask,Vls,Vls,Vp # generate the select mask
vperm  Vmask,Vls,Vlo,Vp # rotate the data
vsel   Vlo,VS,VS,Vp   # insert LSQ component
vsel   Vhi,Vhi,VS,Vmask # insert MSQ component
stvx   Vlo,0,Rb       # store LSQ
addi   Rb,Rb,-16      # address of MSQ
stvx   Vhi,0,Rb       # store MSQ
```
6.5 Vector Integer Operations

Many of the instructions that produce fixed-point integer results have the potential to compute a result value that cannot be represented in the target format. When this occurs, this unrepresentable intermediate value is converted to a representable result value using one of the following methods.

1. The high-order bits of the intermediate result that do not fit in the target format are discarded. This method is used by instructions having names that include the word "Modulo".

2. The intermediate result is converted to the nearest value that is representable in the target format (i.e., to the minimum or maximum representable value, as appropriate). This method is used by instructions having names that include the word "Saturate". An intermediate result that is forced to the minimum or maximum representable value as just described is said to "saturate".

An instruction for which an intermediate result saturates causes $\text{SAT}$ to be set to 1; see Section 6.3.2.

3. If the intermediate result includes non-zero fraction bits it is rounded up to the nearest fixed-point integer value. This method is used by the six Vector Average Integer instructions and by the Vector Multiply-High-Round-Add Signed Halfword Saturate instruction. The latter instruction then uses method 2, if necessary.

---

**Programming Note**

Because $\text{SAT}$ is sticky, it can be used to detect whether any instruction in a sequence of “Saturate”-type instructions produced an inexact result due to saturation. For example, the contents of the VSCR can be copied to a VSR (mfvscr), bits other than $\text{SAT}$ can be cleared in the VSR (vand with a constant), the result can be compared to zero setting CR6 (vcmpequb), and a branch can be taken according to whether $\text{SAT}$ was set to 1 (Branch Conditional that tests CR field 6).

Testing $\text{SAT}$ after each “Saturate”-type instruction would degrade performance considerably. Alternative techniques include the following:

- Retain sufficient information at “checkpoints” that the sequence of computations performed between one checkpoint and the next can be redone (more slowly) in a manner that detects exactly when saturation occurs. Test $\text{SAT}$ only at checkpoints, or when redoing a sequence of computations that saturated.

- Perform intermediate computations using an element length sufficient to prevent saturation, and then use a Vector Pack Integer Saturate instruction to pack the final result to the desired length. (Vector Pack Integer Saturate causes results to saturate if necessary, and sets $\text{SAT}$ to 1 if any result saturates.)
6.5.1 Integer Saturation

Saturation occurs whenever the result of a saturating instruction does not fit in the result field. Unsigned saturation clamps results to zero (0) on underflow and to the maximum positive integer value ($2^{n-1}$, e.g. 255 for byte fields) on overflow. Signed saturation clamps results to the smallest representable negative number ($-2^{n-1}$, e.g. -128 for byte fields) on underflow, and to the largest representable positive number ($2^{n-1}-1$, e.g. +127 for byte fields) on overflow.

In most cases, the simple maximum/minimum saturation performed by the vector instructions is adequate. However, sometimes, e.g. in the creation of very high quality images, more complex saturation functions must be applied. To support this, the Vector facility provides a mechanism for detecting that saturation has occurred. The VSCR has a bit, SAT, which is set to a one (1) anytime any field in a saturating instruction saturates. SAT can only be cleared by explicitly writing zero to it. Thus SAT accumulates a summary result of any integer overflow or underflow that occurs on a saturating instruction.

Borderline cases that generate results equal to saturation values, for example unsigned $0+0=0$ and unsigned byte $1+254=255$, are not considered saturation conditions and do not cause SAT to be set.

SAT can be set by the following types of instructions:

- Move To VSCR
- Vector Add Integer with Saturation
- Vector Subtract Integer with Saturation
- Vector Multiply-Add Integer with Saturation
- Vector Multiply-Sum with Saturation
- Vector Sum-Across with Saturation
- Vector Pack with Saturation
- Vector Convert to Fixed-point with Saturation

Note that only instructions that explicitly call for "saturation" can set SAT. “Modulo” integer instructions and floating-point arithmetic instructions never set SAT.

--- Programming Note ---

The SAT state can be tested and used to alter program flow by moving the VSCR to a VSR (with mfvscr), then masking out bits 0:126 (to clear undefined and reserved bits) and performing a vector compare equal-to unsigned byte w/record (vcmpequb) with zero to get a testable value into the condition register for consumption by a subsequent branch.

Since mfvscr will be slow compared to other Vector instructions, reading and testing SAT after each instruction would be prohibitively expensive. Therefore, software is advised to employ strategies that minimize checking SAT. For example: checking SAT periodically and backtracking to the last checkpoint to identify exactly which field in which instruction saturated; or, working in an element size sufficient to prevent any overflow or underflow during intermediate calculations, then packing down to the desired element size as the final operation (the vector pack instruction saturates the results and updates SAT when a loss of significance is detected).
6.6 Vector Floating-Point Operations

6.6.1 Floating-Point Overview

Unless \( \|J\| = 1 \) (see Section 6.3.2), the floating-point model provided by the Vector Facility conforms to The Java Language Specification (hereafter referred to as “Java”), which is a subset of the default environment specified by the IEEE standard (i.e., by ANSI/IEEE Standard 754-1985, “IEEE Standard for Binary Floating-Point Arithmetic”). For aspects of floating-point behavior that are not defined by Java but are defined by the IEEE standard, vector floating-point conforms to the IEEE standard. For aspects of floating-point behavior that are defined neither by Java nor by the IEEE standard but are defined by the “C9X Floating-Point Proposal” (hereafter referred to as “C9X”), vector floating-point conforms to C9X.

The single-precision floating-point data format, value representations, and computational models defined in Chapter 4, “Floating-Point Facility” on page 131 apply to vector floating-point except as follows.

- In general, no status bits are set to reflect the results of floating-point operations. The only exception is that \( \text{SAT} \) may be set by the Vector Convert To Fixed-Point Word instructions.

- With the exception of the two Vector Convert To Fixed-Point Word instructions and three of the four Vector Round to Floating-Point Integer instructions, all vector floating-point instructions that round use the rounding mode Round to Nearest.

- Floating-point exceptions (see Section 6.6.2) cannot cause the system error handler to be invoked.

Programmed Note

If a function is required that is specified by the IEEE standard, is not supported by the Vector Facility, and cannot be emulated satisfactorily using the functions that are supported by the Vector Facility, the functions provided by the Floating-Point Facility should be used; see Chapter 4.

6.6.2 Floating-Point Exceptions

The following floating-point exceptions may occur during execution of vector floating-point instructions.

- NaN Operand Exception
- Invalid Operation Exception
- Zero Divide Exception
- Log of Zero Exception
- Overflow Exception
- Underflow Exception

If an exception occurs, a result is placed into the corresponding target element as described in the following subsections. This result is the default result specified by Java, the IEEE standard, or C9X, as applicable.

Recall that denormalized source values are treated as if they were zero when \( \|J\| = 1 \). This has the following consequences regarding exceptions.

- Exceptions that can be caused by a zero source value can be caused by a denormalized source value when \( \|J\| = 1 \).

- Exceptions that can be caused by a nonzero source value cannot be caused by a denormalized source value when \( \|J\| = 1 \).

6.6.2.1 NaN Operand Exception

A NaN Operand Exception occurs when a source value for any of the following instructions is a NaN.

- A vector instruction that would normally produce floating-point results
- Either of the two Vector Convert To Fixed-Point Word instructions
- Any of the four Vector Floating-Point Compare instructions

The following actions are taken:

If the vector instruction would normally produce floating-point results, the corresponding result is a source NaN selected as follows. In all cases, if the selected source NaN is a Signaling NaN it is converted to the corresponding Quiet NaN (by setting the high-order bit of the fraction field to 1) before being placed into the target element.
if the element in VSR[VRA+32] is a NaN then the result is that NaN else if the element in VSR[VRB+32] is a NaN then the result is that NaN else if the element in VSR[VRC+32] is a NaN then the result is that NaN else if Invalid Operation exception (Section 6.6.2.2) then the result is the QNaN 0x7FC0_0000

If the instruction is either of the two Vector Convert To Fixed-Point Word instructions, the corresponding result is 0x0000_0000. SAT is not affected.

If the instruction is Vector Compare Bounds Floating-Point, the corresponding result is 0xC000_0000.

If the instruction is one of the other Vector Floating-Point Compare instructions, the corresponding result is 0x0000_0000.

**6.6.2.2 Invalid Operation Exception**

An Invalid Operation Exception occurs when a source value or set of source values is invalid for the specified operation. The invalid operations are:

- Magnitude subtraction of infinities
- Multiplication of infinity by zero
- Reciprocal square root estimate of a negative, nonzero number or -infinity.
- Log base 2 estimate of a negative, nonzero number or -infinity.

The corresponding result is the QNaN 0x7FC0_0000.

**6.6.2.3 Zero Divide Exception**

A Zero Divide Exception occurs when a Vector Reciprocal Estimate Floating-Point or Vector Reciprocal Square Root Estimate Floating-Point instruction is executed with a source value of zero.

The corresponding result is an infinity, where the sign is the sign of the source value.

**6.6.2.4 Log of Zero Exception**

A Log of Zero Exception occurs when a Vector Log Base 2 Estimate Floating-Point instruction is executed with a source value of zero.

The corresponding result is -Infinity.

**6.6.2.5 Overflow Exception**

An Overflow Exception occurs under either of the following conditions.

- For a vector instruction that would normally produce floating-point results, the magnitude of what would have been the result if the exponent range were unbounded exceeds that of the largest finite floating-point number for the target floating-point format.

- For either of the two Vector Convert To Fixed-Point Word instructions, either a source value is an infinity or the product of a source value and $2^{UIM}$ is a number too large in magnitude to be represented in the target fixed-point format.

The following actions are taken:

1. If the vector instruction would normally produce floating-point results, the corresponding result is an infinity, where the sign is the sign of the intermediate result.

2. If the instruction is Vector Convert To Unsigned Fixed-Point Word Saturate, the corresponding result is 0xFFFF_FFFF if the source value is a positive number or +infinity, and is 0x0000_0000 if the source value is a negative number or -infinity. SAT is set to 1.

3. If the instruction is Vector Convert To Signed Fixed-Point Word Saturate, the corresponding result is 0x7FFF_FFFF if the source value is a positive number or +infinity, and is 0x8000_0000 if the source value is a negative number or -infinity. SAT is set to 1.

**6.6.2.6 Underflow Exception**

An Underflow Exception can occur only for vector instructions that would normally produce floating-point results. It is detected before rounding. It occurs when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded is less in magnitude than the smallest normalized floating-point number for the target floating-point format.

The following actions are taken:

1. If $NJ = 0$, the corresponding result is the value produced by denormalizing and rounding the intermediate result.

2. If $NJ = 1$, the corresponding result is a zero, where the sign is the sign of the intermediate result.
6.7 Vector Storage Access Instructions

The Vector Storage Access instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.10.3, “Effective Address Calculation” on page 29. The low-order bits of the EA that would correspond to an unaligned storage operand are ignored.

The Load Vector Element Indexed and Store Vector Element Indexed instructions transfer a byte, halfword, or word element between storage and a VSR. The Load Vector Indexed and Store Vector Indexed instructions transfer an aligned quadword between storage and a VSR.

6.7.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (Store only), or if the program attempts to access storage that is unavailable.
6.7.2 Vector Load Instructions

The aligned byte, halfword, word, or quadword in storage addressed by EA is loaded into VSR[VRT+32].

Programming Note

The Load Vector Element instructions load the specified element into the same location in the target register as the location into which it would be loaded using the Load Vector instruction.

Load Vector Element Byte Indexed X-form

\texttt{lvebx VRT,RA,RB}

if MSR.VEC=0 then Vector_Unavailable()

\[EA \leftarrow \begin{cases} \text{GPR}[RA] & \text{if RA} \neq 0 \\ 0 & \text{otherwise} \end{cases} + \text{GPR}[RB] \]

\[eb \leftarrow \text{EA}.\text{bit}[60:63] \]

\[\text{VSR}[VRT+32] \leftarrow \text{undefined} \]

if Big-Endian byte ordering then

\[\text{VSR}[VRT+32].\text{byte}[eb] \leftarrow \text{MEM}(EA,1) \]

else

\[\text{VSR}[VRT+32].\text{byte}[15-eb] \leftarrow \text{MEM}(EA,1) \]

Let \(EA\) be the sum of the contents of GPR[RA], or 0 if \(RA=0\), and the contents of GPR[RB].

Let \(eb\) be bits 60:63 of \(EA\).

If Big-Endian byte ordering is used for the storage access, the contents of the byte in storage at address \(EA\) are placed into byte \(eb\) of VSR[VRT+32]. The remaining bytes of VSR[VRT+32] are set to undefined values.

If Little-Endian byte ordering is used for the storage access, the contents of the byte in storage at address \(EA\) are placed into byte 15-\(eb\) of VSR[VRT+32]. The remaining bytes of VSR[VRT+32] are set to undefined values.

Special Registers Altered:

None

Register Data Layout for lvebx

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR[RA]</td>
<td>GPR[RB]</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

0 63 127
Load Vector Element Halfword Indexed X-form

lvehx VRT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>VRT</th>
<th>RA</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

EA ← EA & 0xFFFF_FFFF_FFFF_FFFE

eb ← EA.bit[60:63]

VSR[VRT+32] ← undefined
if Big-Endian byte ordering then

VSR[VRT+32].byte[eb:eb+1] ← MEM(EA, 2)
else

VSR[VRT+32].byte[14-eb:15-eb] ← MEM(EA, 2)

Let EA be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE
with the sum of the contents of GPR[RA], or 0 if RA=0,
and the contents of GPR[RB].

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

– the contents of the byte in storage at address EA
  are placed into byte eb of VSR[VRT+32],
– the contents of the byte in storage at address EA+1
  are placed into byte eb+1 of VSR[VRT+32], and
– the remaining bytes of VSR[VRT+32] are set to
  undefined values.

If Little-Endian byte ordering is used for the storage access,

– the contents of the byte in storage at address EA
  are placed into byte 15-eb of VSR[VRT+32],
– the contents of the byte in storage at address EA+1
  are placed into byte 14-eb of VSR[VRT+32], and
– the remaining bytes of VSR[VRT+32] are set to
  undefined values.

Special Registers Altered:

None

Register Data Layout for lvehx

src1 GPR[RA]
src2 GPR[RB]
result VSR[VRT+32]
Load Vector Element Word Indexed X-form

\[ \text{lvewx} \quad \text{VRT,RA,RB} \]

\[
\begin{array}{cccc|c}
0 & 31 & 6 & 11 & 16 & 21 & 60\ldots & 71 \\
\end{array}
\]

If MSR.VEC=0 then Vector_Unavailable()

\[ \text{EA} \leftarrow \begin{cases}
(\text{RA}=0) \? 0 & \text{GPR}[\text{RA}] + \text{GPR}[\text{RB}] \\
\text{EA} & \text{if Big-Endian byte ordering then}
\end{cases} \]

\[ \text{eb} \leftarrow \text{EA}.\text{bit}[60:63] \]

\[ \text{VSR}[\text{VRT}+32] \leftarrow \text{undefined} \]

\[ \begin{cases}
\text{if Big-Endian byte ordering then} & \\
\text{else} & \\
\end{cases}
\]

\[ \text{VSR}[\text{VRT}+32].\text{byte}[\text{eb}:\text{eb}+3] \leftarrow \text{MEM}(\text{EA}, 4) \]

Let \( \text{EA} \) be the result of ANDing \( 0xFFFF_FFFF_FFFF_FFFC \) with the sum of the contents of \( \text{GPR}[\text{RA}] \), or 0 if \( \text{RA}=0 \), and the contents of \( \text{GPR}[\text{RB}] \).

Let \( \text{eb} \) be bits 60:63 of \( \text{EA} \).

If Big-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address \( \text{EA} \) are placed into byte 15-eb of \( \text{VSR}[\text{VRT}+32] \),
- the contents of the byte in storage at address \( \text{EA}+1 \) are placed into byte 14-eb of \( \text{VSR}[\text{VRT}+32] \),
- the contents of the byte in storage at address \( \text{EA}+2 \) are placed into byte 13-eb of \( \text{VSR}[\text{VRT}+32] \),
- the contents of the byte in storage at address \( \text{EA}+3 \) are placed into byte 12-eb of \( \text{VSR}[\text{VRT}+32] \), and
- the remaining bytes of \( \text{VSR}[\text{VRT}+32] \) are set to undefined values.

Special Registers Altered:
None

Register Data Layout for lvewx

\[
\begin{array}{cccc}
\text{src}1 & \text{GPR}[\text{RA}] \\
\text{src}2 & \text{GPR}[\text{RB}] \\
\text{result} & \text{VSR}[\text{VRT}+32] \\
0 & 63 & 127
\end{array}
\]

If Little-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address \( \text{EA} \) are placed into byte 15-eb of \( \text{VSR}[\text{VRT}+32] \),
- the contents of the byte in storage at address \( \text{EA}+1 \) are placed into byte 14-eb of \( \text{VSR}[\text{VRT}+32] \),
- the contents of the byte in storage at address \( \text{EA}+2 \) are placed into byte 13-eb of \( \text{VSR}[\text{VRT}+32] \),
- the contents of the byte in storage at address \( \text{EA}+3 \) are placed into byte 12-eb of \( \text{VSR}[\text{VRT}+32] \), and
- the remaining bytes of \( \text{VSR}[\text{VRT}+32] \) are set to undefined values.
Load Vector Indexed X-form

\texttt{lvx VRT,RA,RB}

\begin{verbatim}
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
EA ← EA & 0xFFFF_FFFF_FFFF_FFF0
VSR[VRT+32] ← MEM(EA, 16)
\end{verbatim}

Let \( EA \) be the result of ANDing \( 0xFFFF_FFFF_FFFF_FFF0 \) with the sum of the contents of \( GPR[RA] \), or 0 if \( RA=0 \), and the contents of \( GPR[RB] \).

The contents of the quadword in storage at address \( EA \) are placed into \( VSR[VRT+32] \).

Special Registers Altered:
None

Load Vector Indexed Last X-form

\texttt{lvxl VRT,RA,RB}

\begin{verbatim}
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
EA ← EA & 0xFFFF_FFFF_FFFF_FFF0
VSR[VRT+32] ← MEM(EA, 16)
mark_as_not_likely_to_be_needed_again_anytime_soon(EA)
\end{verbatim}

Let \( EA \) be the result of ANDing \( 0xFFFF_FFFF_FFFF_FFF0 \) with the sum of the contents of \( GPR[RA] \), or 0 if \( RA=0 \), and the contents of \( GPR[RB] \).

The contents of the quadword in storage at address \( EA \) are placed into \( VSR[VRT+32] \).

\texttt{lvxl} provides a hint that the quadword in storage addressed by \( EA \) will probably not be needed again by the program in the near future.

Special Registers Altered:
None

Register Data Layout for \texttt{lxvx} & \texttt{lxvl}

\begin{verbatim}
src1src2result
\hline
GPR[RA]GPR[RB]VSR[VRT+32]
\end{verbatim}

Programming Note

On some implementations, the hint provided by the \texttt{lvxl} instruction and the corresponding hint provided by the \texttt{stvxl} instruction are applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for replacement when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference.
6.7.3 Vector Store Instructions

Some portion or all of the contents of $VSR\{VRS+32\}$ are stored into the aligned byte, halfword, word, or quadword in storage addressed by $EA$.

Programming Note

The Store Vector Element instructions store the specified element into the same storage location as the location into which it would be stored using the Store Vector instruction.

Store Vector Element Byte Indexed X-form

$$\text{stvebx VRS,RA,RB}$$

1. If $VSR.VEC=0$ then $\text{Vector\_Unavailable()}$

   $$EA \leftarrow ((RA=0) ? 0 : \text{GPR}[RA]) + \text{GPR}[RB]$$

   $$eb \leftarrow EA.\text{bit}[60:63]$$

2. If Big-Endian byte ordering then

   $$\text{MEM}(EA,1) \leftarrow VSR\{VRS+32\}.\text{byte}[eb]$$

3. Else

   $$\text{MEM}(EA,1) \leftarrow VSR\{VRS+32\}.\text{byte}[15-eb]$$

Let $EA$ be the sum of the contents of $\text{GPR}[RA]$, or 0 if $RA=0$, and the contents of $\text{GPR}[RB]$.

Let $eb$ be bits $60:63$ of $EA$.

If Big-Endian byte ordering is used for the storage access, the contents of byte $eb$ of $VSR\{VRS+32\}$ are placed in the byte in storage at address $EA$.

If Little-Endian byte ordering is used for the storage access, the contents of byte $15-eb$ of $VSR\{VRS+32\}$ are placed in the byte in storage at address $EA$.

Special Registers Altered:

None

Programming Note

Unless bits $60:63$ of the address are known to match the byte offset of the subject byte element in $VSR\{VRS+32\}$, software should use $\text{Vector Splat}$ to splat the subject byte element before performing the store.

Register Data Layout for stvebx

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR[RA]</td>
<td>GPR[RB]</td>
<td>VSR[VRS+32]</td>
</tr>
</tbody>
</table>

0 | 63 | 127 |
Store Vector Element Halfword Indexed X-form

stvehx VRS,RA,RB

if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
EA ← EA & 0xFFFF_FFFF_FFFF_FFFE
eb ← EA.bit[60:63]

if Big-Endian byte ordering then
    MEM(EA,2) ← VSR[VRS+32].byte[eb:eb+1]
else
    MEM(EA,2) ← VSR[VRS+32][14-eb:15-eb]

Let EA be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE
with the sum of the contents of GPR[RA], or 0 if RA=0,
and the contents of GPR[RB].

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

– the contents of byte eb of VSR[VRS+32] are placed in the byte in storage at address EA, and
– the contents of byte eb+1 of VSR[VRS+32] are placed in the byte in storage at address EA+1.

If Little-Endian byte ordering is used for the storage access,

– the contents of byte 15-eb of VSR[VRS+32] are placed in the byte in storage at address EA, and
– the contents of byte 14-eb of VSR[VRS+32] are placed in the byte in storage at address EA+1.

Special Registers Altered:
None

Programming Note

Unless bits 60:62 of the address are known to match the halfword offset of the subject halfword element in VSR[VRS+32] software should use Vector Splat to splat the subject halfword element before performing the store.

Register Data Layout for stvehx

```
src1 GPR[RA]
src2 GPR[RB]
src3 VSR[VRS+32]
```

Chapter 6. Vector Facility  273
**Store Vector Element Word Indexed X-form**

### stvewx VRS,RA,RB

| 31 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | VA |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

- if MSR.VEC=0 then Vector_Unavailable();
- EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
- eb ← EA.bit[60:63]

if Big-Endian byte ordering then
    MEM(EA,4) ← VSR[VRS+32].byte[eb:eb+3]
else
    MEM(EA,4) ← VSR[VRS+32].byte[12-eb:15-eb]

Let EA be the result of ANDing 0xFFFF_FFFF_FFFF_FFFC with the sum of the contents of GPR[RA], or 0 if RA = 0, and the contents of GPR[RB].

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of byte 15-eb of VSR[VRS+32] are placed in the byte in storage at address EA,
- the contents of byte 14-eb of VSR[VRS+32] are placed in the byte in storage at address EA+1,
- the contents of byte 13-eb of VSR[VRS+32] are placed in the byte in storage at address EA+2, and
- the contents of byte 12-eb of VSR[VRS+32] are placed in the byte in storage at address EA+3.

Special Registers Altered:
- None

**Programming Note**

Unless bits 60:61 of the address are known to match the word offset of the subject word element in VSR[VRS+32], software should use Vector Splat to splat the subject word element before performing the store.

### Register Data Layout for stvewx

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR[RA]</td>
<td>GPR[RB]</td>
<td>VSR[VRS+32]</td>
</tr>
</tbody>
</table>

If Little-Endian byte ordering is used for the storage access,

- the contents of byte 15-eb of VSR[VRS+32] are placed in the byte in storage at address EA,
- the contents of byte 14-eb of VSR[VRS+32] are placed in the byte in storage at address EA+1,
- the contents of byte 13-eb of VSR[VRS+32] are placed in the byte in storage at address EA+2, and
- the contents of byte 12-eb of VSR[VRS+32] are placed in the byte in storage at address EA+3.
**Store Vector Indexed X-form**

\[ \text{stvx } \text{VRS,RA,RB} \]

if MSR.VEC=0 then Vector_Unavailable();

\[ EA \leftarrow (\text{RA}=0 ? 0 : \text{GPR}[\text{RA}]) + \text{GPR}[\text{RB}] \]

\[ EA \leftarrow EA \& \text{0xFFFF_FFFF_FFFF_FFF0} \]

\[ \text{MEM}(EA, 16) \leftarrow \text{VSR}[\text{VRS+32}] \]

Let \( EA \) be the result of ANDing \text{0xFFFF_FFFF_FFFF_FFF0} with the sum of the contents of \( \text{GPR}[\text{RA}], \) or \( 0 \) if \( \text{RA}=0 \), and the contents of \( \text{GPR}[\text{RB}] \).

The contents of \( \text{VSR}[\text{VRS+32}] \) are placed into the quadword in storage at address \( EA \).

**Special Registers Altered:**

None

---

**Store Vector Indexed Last X-form**

\[ \text{stvl } \text{VRS,RA,RB} \]

if MSR.VEC=0 then Vector_Unavailable();

\[ EA \leftarrow (\text{RA}=0 ? 0 : \text{GPR}[\text{RA}]) + \text{GPR}[\text{RB}] \]

\[ EA \leftarrow EA \& \text{0xFFFF_FFFF_FFFF_FFF0} \]

\[ \text{MEM}(EA, 16) \leftarrow \text{VSR}[\text{VRS+32}] \]

\[ \text{mark_as_not_likely_to_be_needed_again_anytime_soon}(EA) \]

Let \( EA \) be the result of ANDing \text{0xFFFF_FFFF_FFFF_FFF0} with the sum of the contents of \( \text{GPR}[\text{RA}], \) or \( 0 \) if \( \text{RA}=0 \), and the contents of \( \text{GPR}[\text{RB}] \).

The contents of \( \text{VSR}[\text{VRS+32}] \) are placed into the quadword in storage at address \( EA \).

\text{stvl} provides a hint that the quadword in storage addressed by \( EA \) will probably not be needed again by the program in the near future.

**Special Registers Altered:**

None

---

**Register Data Layout for stvx & stvl**

<table>
<thead>
<tr>
<th>src1</th>
<th>GPR[RA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>GPR[RB]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[VRS+32]</td>
</tr>
</tbody>
</table>

See the Programming Note for the \text{lvxl} instruction on page 271.
6.7.4 Vector Alignment Support Instructions

Programming Note

The `lvsl` and `lvsr` instructions can be used to create the permute control vector to be used by a subsequent `vperm` instruction (see page 296). Let \( X \) and \( Y \) be the contents of \( \text{VSR}[\text{VRA}+32] \) and \( \text{VSR}[\text{VRB}+32] \) specified by the `vperm`. The control vector created by `lvsl` causes the `vperm` to select the high-order 16 bytes of the result of shifting the 32-byte value \( X || Y \) left by \( s_1 \) bytes. The control vector created by `lvsr` causes the `vperm` to select the low-order 16 bytes of the result of shifting \( X || Y \) right by \( s_2 \) bytes.

Programming Note

Examples of uses of `lvsl`, `lvsr`, and `vperm` to load and store unaligned data are given in Section 6.4.1. These instructions can also be used to rotate or shift the contents of a VSR left (`lvsl`) or right (`lvsr`) by \( s_1 \) bytes. For rotating, the VSR to be rotated should be specified as both \( \text{VSR}[\text{VRA}+32] \) and \( \text{VSR}[\text{VRB}+32] \) for `vperm`. For shifting left, \( \text{VSR}[\text{VRB}+32] \) for `vperm` should be a register containing all zeros and \( \text{VSR}[\text{VRA}+32] \) should contain the value to be shifted, and vice versa for shifting right.
Load Vector for Shift Left Indexed X-form

```plaintext
lvsl VRT,RA,RB

if MSR.VEC=0 then Vector_Unavailable()

sh ← (((RA=0) ? 0 : GPR[RA]) + GPR[RB]).bit[60:63]

switch(sh)

  case(0x0): VSR[VRT+32] ← 0x000102030405060708090A0B0C0D0E0F
  case(0x1): VSR[VRT+32] ← 0x0102030405060708090A0B0C0D0E0F10
  case(0x2): VSR[VRT+32] ← 0x02030405060708090A0B0C0D0E0F11
  case(0x3): VSR[VRT+32] ← 0x030405060708090A0B0C0D0E0F1112
  case(0x4): VSR[VRT+32] ← 0x0405060708090A0B0C0D0E0F101112
  case(0x5): VSR[VRT+32] ← 0x05060708090A0B0C0D0E0F1011121314
  case(0x6): VSR[VRT+32] ← 0x060708090A0B0C0D0E0F101112131415
  case(0x7): VSR[VRT+32] ← 0x0708090A0B0C0D0E0F10111213141516
  case(0x8): VSR[VRT+32] ← 0x08090A0B0C0D0E0F1011121314151617
  case(0x9): VSR[VRT+32] ← 0x090A0B0C0D0E0F101112131415161718
  case(0xA): VSR[VRT+32] ← 0x0A0B0C0D0E0F10111213141516171819
  case(0xB): VSR[VRT+32] ← 0x0B0C0D0E0F101112131415161718191A
  case(0xC): VSR[VRT+32] ← 0x0C0D0E0F101112131415161718191A1B
  case(0xD): VSR[VRT+32] ← 0x0D0E0F101112131415161718191A1B1C
  case(0xE): VSR[VRT+32] ← 0x0E0F101112131415161718191A1B1C1D
  case(0xF): VSR[VRT+32] ← 0x0F101112131415161718191A1B1C1D1E
```

Let `sh` be bits 60:63 of the sum of the contents of `GPR[RA]`, or 0 if `RA=0`, and the contents of `GPR[RB]`.

Let `X` be the 32-byte value 0x00 || 0x01 || 0x02 || ... || 0x1D || 0x1E || 0x1F.

Bytes `sh` to `sh+15` of `X` are placed into `VSR[VRT+32]`.

Special Registers Altered:
None

---

Load Vector for Shift Right Indexed X-form

```plaintext
lvsr VRT,RA,RB

if MSR.VEC=0 then Vector_Unavailable()

sh ← (((RA=0) ? 0 : GPR[RA]) + GPR[RB]).bit[60:63]

switch(sh)

  case(0x0): VSR[VRT+32] ← 0x101112131415161718191A1B1C1D1E1F
  case(0x1): VSR[VRT+32] ← 0x0F101112131415161718191A1B1C1D1E
  case(0x2): VSR[VRT+32] ← 0x0E0F101112131415161718191A1B1C1D
  case(0x3): VSR[VRT+32] ← 0x0D0E0F101112131415161718191A1B1C
  case(0x4): VSR[VRT+32] ← 0x0C0D0E0F101112131415161718191A1B
  case(0x5): VSR[VRT+32] ← 0x0B0C0D0E0F101112131415161718191A
  case(0x6): VSR[VRT+32] ← 0x0A0B0C0D0E0F10111213141516171819
  case(0x7): VSR[VRT+32] ← 0x090A0B0C0D0E0F1011121314151617
  case(0x8): VSR[VRT+32] ← 0x08090A0B0C0D0E0F10111213141516
  case(0x9): VSR[VRT+32] ← 0x0708090A0B0C0D0E0F101112131415
  case(0xA): VSR[VRT+32] ← 0x060708090A0B0C0D0E0F1011121314
  case(0xB): VSR[VRT+32] ← 0x05060708090A0B0C0D0E0F10111213
  case(0xC): VSR[VRT+32] ← 0x0405060708090A0B0C0D0E0F10111213
  case(0xD): VSR[VRT+32] ← 0x030405060708090A0B0C0D0E0F10111213
  case(0xE): VSR[VRT+32] ← 0x02030405060708090A0B0C0D0E0F10111213
  case(0xF): VSR[VRT+32] ← 0x0102030405060708090A0B0C0D0E0F10
```

Let `sh` be bits 60:63 of the sum of the contents of `GPR[RA]`, or 0 if `RA=0`, and the contents of `GPR[RB]`.

Let `X` be the 32-byte value 0x00 || 0x01 || 0x02 || ... || 0x1D || 0x1E || 0x1F.

Bytes `16-sh` to `31-sh` of `X` are placed into `VSR[VRT+32]`.

Special Registers Altered:
None

---

Register Data Layout for lvsl & lvsr

<table>
<thead>
<tr>
<th>src1</th>
<th>GPR[RA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>GPR[RB]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
<td>127</td>
</tr>
</tbody>
</table>
6.8 Vector Permute and Formatting Instructions

6.8.1 Vector Pack Instructions

**Vector Pack Pixel Vx-form**

**vpkpx**

\[
\begin{array}{cccc}
4 & VRT & 11 & 16 & 21 & 782 \\
0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

- Let \( \text{vsrc} \) be the concatenation of the contents of \( \text{VSR}[\text{VRA}+32] \) followed by the contents of \( \text{VSR}[\text{VRB}+32] \).
- For each integer value \( i \) from 0 to 7, do the following.

  - The contents of word element \( i \) of \( \text{vsrc} \) are packed to produce a 16-bit value as described below.

    - \( \text{bit} 7 \) of the first byte (\( \text{bit} 7 \) of the word)
    - \( \text{bits} 0:4 \) of the second byte (\( \text{bits} 8:12 \) of the word)
    - \( \text{bits} 0:4 \) of the third byte (\( \text{bits} 16:20 \) of the word)
    - \( \text{bits} 0:4 \) of the fourth byte (\( \text{bits} 24:28 \) of the word)

  - The result is placed into halfword element \( i \) of \( \text{VSR}[\text{VRT}+32] \).

**Special Registers Altered:**

None

**Register Data Layout for vpkpx**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[\text{VRA}+32].word[0]</th>
<th>VSR[\text{VRA}+32].word[1]</th>
<th>VSR[\text{VRA}+32].word[2]</th>
<th>VSR[\text{VRA}+32].word[3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[\text{VRB}+32].word[0]</td>
<td>VSR[\text{VRB}+32].word[1]</td>
<td>VSR[\text{VRB}+32].word[2]</td>
<td>VSR[\text{VRB}+32].word[3]</td>
</tr>
</tbody>
</table>

**Programming Note**

Each source word can be considered to be a 32-bit "pixel", consisting of four 8-bit "channels". Each target halfword can be considered to be a 16-bit pixel, consisting of one 1-bit channel and three 5-bit channels. A channel can be used to specify the intensity of a particular color, such as red, green, or blue, or to provide other information needed by the application.
Vector Pack Signed Halfword Signed Saturate VX-form

\[ \text{vpkshss} \rightarrow VRT, VRA, VRB \]

If MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{vsrc}.\text{qword}[0] &\leftarrow \text{VSR}[\text{VRA}+32] \\
\text{vsrc}.\text{qword}[1] &\leftarrow \text{VSR}[\text{VRB}+32] \\
\end{align*}
\]

\[
\begin{align*}
do\ i &\leftarrow 0\ to\ 15 \\
\text{VSR}[\text{VRT}+32].\text{byte}[i] &\leftarrow \text{si8_CLAMP}(\text{EXTS}\text{vsrc}.\text{hword}[i]) \\
\end{align*}
\]

Let vsrc be the concatenation of the contents of VSR[VRT+32] followed by the contents of VSR[VRB+32].

For each integer value \( i \) from 0 to 15, do the following.

The signed integer value in halfword element \( i \) of vsrc is placed into byte element \( i \) of VSR[VRT+32] in signed integer format.

- If the value of the element is greater than \( 2^7-1 \) the result saturates to \( 2^7-1 \) and SAT is set to 1.
- If the value of the element is less than \( -2^7 \) the result saturates to \( -2^7 \) and SAT is set to 1.

Special Registers Altered:

SAT

Vector Pack Signed Halfword Unsigned Saturate VX-form

\[ \text{vpkhus} \rightarrow VRT, VRA, VRB \]

If MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{vsrc}.\text{qword}[0] &\leftarrow \text{VSR}[\text{VRA}+32] \\
\text{vsrc}.\text{qword}[1] &\leftarrow \text{VSR}[\text{VRB}+32] \\
\end{align*}
\]

\[
\begin{align*}
do\ i &\leftarrow 0\ to\ 15 \\
\text{VSR}[\text{VRT}+32].\text{byte}[i] &\leftarrow \text{ui8_CLAMP}(\text{EXTS}\text{vsrc}.\text{hword}[i]) \\
\end{align*}
\]

Let vsrc be the concatenation of the contents of VSR[VRT+32] followed by the contents of VSR[VRB+32].

For each integer value \( i \) from 0 to 15, do the following.

The signed integer value in halfword element \( i \) of vsrc is placed into byte element \( i \) of VSR[VRT+32] in unsigned integer format.

- If the value of the element is greater than \( 2^8-1 \) the result saturates to \( 2^8-1 \) and SAT is set to 1.
- If the value of the element is less than 0 the result saturates to 0 and SAT is set to 1.

Special Registers Altered:

SAT

Register Data Layout for vpkshss & vpkhus

\[
\begin{array}{cccccccccccccccccccc}
\text{src1} & \text{VSR}[\text{VRA}+32].\text{hword}[0] & \text{VSR}[\text{VRA}+32].\text{hword}[1] & \text{VSR}[\text{VRA}+32].\text{hword}[2] & \text{VSR}[\text{VRA}+32].\text{hword}[3] & \text{VSR}[\text{VRA}+32].\text{hword}[4] & \text{VSR}[\text{VRA}+32].\text{hword}[5] & \text{VSR}[\text{VRA}+32].\text{hword}[6] & \text{VSR}[\text{VRA}+32].\text{hword}[7] \\
\text{src2} & \text{VSR}[\text{VRB}+32].\text{hword}[0] & \text{VSR}[\text{VRB}+32].\text{hword}[1] & \text{VSR}[\text{VRB}+32].\text{hword}[2] & \text{VSR}[\text{VRB}+32].\text{hword}[3] & \text{VSR}[\text{VRB}+32].\text{hword}[4] & \text{VSR}[\text{VRB}+32].\text{hword}[5] & \text{VSR}[\text{VRB}+32].\text{hword}[6] & \text{VSR}[\text{VRB}+32].\text{hword}[7] \\
\end{array}
\]
Vector Pack Signed Word Signed Saturate VX-form

\[ \text{vpkswss VRT, VRA, VRB} \]

\[
\begin{array}{cccccc}
\text{0} & \text{4} & \text{6} & \text{11} & \text{16} & \text{21} & \text{462} & \text{31}
\end{array}
\]

- \text{if MSR.VEC=0 then VectorUnavailable(\text{\()})}
- \text{vsrc.qword}[0] \leftarrow \text{VSR[VRA+32]}
- \text{vsrc.qword}[1] \leftarrow \text{VSR[VRB+32]}
- \text{do i = 0 to 7}
- \quad \text{VSR[VRT+32].hword}[i] \leftarrow \text{si16.Clamp(EXTS(vsrc.word)[i])}
- \text{end}

\text{Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].}

For each integer value \(i\) from 0 to 7, do the following.
The signed integer value in word element \(i\) of \text{vsrc} is placed into halfword element \(i\) of \text{VSR}[VRT+32] in signed integer format.

- If the value of the element is greater than \(2^{15} - 1\), the result saturates to \(2^{15} - 1\) and SAT is set to 1.
- If the value of the element is less than \(-2^{15}\) the result saturates to \(-2^{15}\) and SAT is set to 1.

Special Registers Altered:
SAT

Register Data Layout for vpkswss & vpkswus

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
<td>64</td>
</tr>
<tr>
<td>60</td>
<td>96</td>
<td>112</td>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>

Vector Pack Signed Word Unsigned Saturate VX-form

\[ \text{vpkswus VRT, VRA, VRB} \]

\[
\begin{array}{cccccc}
\text{0} & \text{4} & \text{6} & \text{11} & \text{16} & \text{21} & \text{334} & \text{31}
\end{array}
\]

- \text{if MSR.VEC=0 then VectorUnavailable(\text{\()})}
- \text{vsrc.qword}[0] \leftarrow \text{VSR[VRA+32]}
- \text{vsrc.qword}[1] \leftarrow \text{VSR[VRB+32]}
- \text{do i = 0 to 7}
- \quad \text{VSR[VRT+32].hword}[i] \leftarrow \text{ui16.Clamp(EXTS(vsrc.word)[i])}
- \text{end}

\text{Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].}

For each integer value \(i\) from 0 to 7, do the following.
The signed integer value in word element \(i\) of \text{vsrc} is placed into halfword element \(i\) of \text{VSR}[VRT+32] in unsigned integer format.

- If the value of the element is greater than \(2^{16} - 1\), the result saturates to \(2^{16} - 1\) and SAT is set to 1.
- If the value of the element is less than 0 the result saturates to 0 and SAT is set to 1.

Special Registers Altered:
SAT
**Vector Pack Signed Doubleword Signed Saturate VX-form**

```
vpksdss  VRT,VRA,VRB

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1486</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ← si32_CLAMP(EXTS(VSR[VRA+32].dword[0]))
VSR[VRT+32].word[1] ← si32_CLAMP(EXTS(VSR[VRA+32].dword[1]))
VSR[VRT+32].word[2] ← si32_CLAMP(EXTS(VSR[VRB+32].dword[0]))
VSR[VRT+32].word[3] ← si32_CLAMP(EXTS(VSR[VRB+32].dword[1]))
```

Let `vsrc` be the concatenation of the contents of `VSR[VRA+32]` followed by the contents of `VSR[VRB+32]`.

For each integer value `i` from 0 to 3, do the following.

The signed integer value in doubleword element `i` of `vsrc` is placed into word element `i` of `VSR[VRT+32]` in signed integer format.

- If the value is greater than `2^{31}-1` the result saturates to `2^{31}-1` and SAT is set to 1.
- If the value is less than `-2^{31}` the result saturates to `-2^{31}` and SAT is set to 1.

**Special Registers Altered:**

SAT

**Register Data Layout for vpksdss & vpksdus**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].word[0]</td>
<td>VSR[VRT+32].word[1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSR[VRT+32].word[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSR[VRT+32].word[3]</td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>96</td>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>

**Vector Pack Signed Doubleword Unsigned Saturate VX-form**

```
vpksdus VRT,VRA,VRB

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1358</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ← ui32.Clamp(EXTS(VSR[VRA+32].dword[0]))
VSR[VRT+32].word[1] ← ui32.Clamp(EXTS(VSR[VRA+32].dword[1]))
VSR[VRT+32].word[2] ← ui32.Clamp(EXTS(VSR[VRB+32].dword[0]))
```

Let `vsrc` be the concatenation of the contents of `VSR[VRA+32]` followed by the contents of `VSR[VRB+32]`.

For each integer value `i` from 0 to 3, do the following.

The signed integer value in doubleword element `i` of `vsrc` is placed into word element `i` of `VSR[VRT+32]` in unsigned integer format.

- If the value is greater than `2^{32}-1` the result saturates to `2^{32}-1` and SAT is set to 1.
- If the value is less than 0 the result saturates to 0 and SAT is set to 1.

**Special Registers Altered:**

SAT
Vector Pack Unsigned Halfword Unsigned Modulo VX-form

vpkuhum VRT,VRA,VRB

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 15
    VSR[VRT+32].byte[i] ← vsrc.hword[i].bit[8:15]
end

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 15, do the following.
The contents of bits 8:15 of halfword element i of vsrc are placed into byte element i of VSR[VRT+32].

Special Registers Altered:
None

Vector Pack Unsigned Halfword Unsigned Saturate VX-form

vpkuhus VRT,VRA,VRB

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 15
    VSR[VRT+32].byte[i] ← ui8_CLAMP(EXTZ(vsrc.hword[i]))
end

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 15, do the following.
The unsigned integer value in halfword element i of vsrc are placed into byte element i of VSR[VRT+32] in unsigned integer format.

- If the value of the element is greater than 2^8 - 1 the result saturates to 2^8 - 1 and SAT is set to 1.

Special Registers Altered:
SAT

Register Data Layout for vpkuhum & vpkuhus

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
</tr>
</tbody>
</table>
Vector Pack Unsigned Word Unsigned Modulo VX-form

\[ \text{vpkuwum} \quad \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC = 0 then Vector_Unavailable()

\[ \text{vsrc.qword[0]} \leftarrow \text{VSR[VRA+32]} \]
\[ \text{vsrc.qword[1]} \leftarrow \text{VSR[VRB+32]} \]

do i = 0 to 7
\[ \text{VSR[VRT+32].hword[i]} \leftarrow \text{vsrc.word[i].bit[16:31]} \]
end

Let \text{vsrc} be the concatenation of the contents of \text{VSR[VRA+32]} followed by the contents of \text{VSR[VRB+32]}.

For each integer value \(i\) from 0 to 7, do the following.

The contents of bits 16:31 of word element \(i\) of \text{vsrc} are placed into halfword element \(i\) of \text{VSR[VRT+32]}.

Special Registers Altered:

None

Vector Pack Unsigned Word Unsigned Saturate VX-form

\[ \text{vpkuwus} \quad \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC = 0 then Vector_Unavailable()

\[ \text{vsrc.qword[0]} \leftarrow \text{VSR[VRA+32]} \]
\[ \text{vsrc.qword[1]} \leftarrow \text{VSR[VRB+32]} \]

do i = 0 to 7
\[ \text{VSR[VRT+32].hword[i]} \leftarrow \text{ui16_CLAMP(EXTZ(vsrc.word[i]))} \]
end

Let \text{vsrc} be the concatenation of the contents of \text{VSR[VRA+32]} followed by the contents of \text{VSR[VRB+32]}.

For each integer value \(i\) from 0 to 7, do the following.

The unsigned integer value in word element \(i\) of \text{vsrc} is placed into halfword element \(i\) of \text{VSR[VRT+32]} in unsigned integer format.

– If the value of the element is greater than \(2^{16} - 1\) the result saturates to \(2^{16} - 1\) and SAT is set to 1.

Special Registers Altered:

SAT

Register Data Layout for \text{vpkuwum} & \text{vpkuwus}

|------|--------------------|--------------------|--------------------|--------------------|

| 0 | 16 | 32 | 48 | 64 | 80 | 96 | 112 | 127 |
Vector Pack Unsigned Doubleword Unsigned Modulo VX-form

vpkudum  VRT,VRA,VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC then Vector_Unavailable()

VSR[VRT+32].word[0] ← VSR[VRA+32].dword[0].bit[32:63]

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 3, do the following.
The contents of bits 32:63 of doubleword element i of vsrc are placed into word element i of VSR[VRT+32].

Special Registers Altered:
None

Vector Pack Unsigned Doubleword Unsigned Saturate VX-form

vpkudus  VRT,VRA,VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC then Vector_Unavailable()

VSR[VRT+32].word[0] ← ui32.Clamp(EXTZ(VSR[VRA+32].dword[0]))
VSR[VRT+32].word[1] ← ui32.Clamp(EXTZ(VSR[VRA+32].dword[1]))
VSR[VRT+32].word[2] ← ui32.Clamp(EXTZ(VSR[VRB+32].dword[0]))

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 3, do the following.
The unsigned integer value in doubleword element i of vsrc are placed into halfword element i of VSR[VRT+32] in unsigned integer format.
– If the value of the element is greater than 2^32-1 the result saturates to 2^32-1 and SAT is set to 1.

Special Registers Altered:
SAT

Register Data Layout for vpkudum & vpkudus

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>
6.8.2 Vector Unpack Instructions

**Vector Unpack High Signed Byte VX-form**

vupkhsb VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>526</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 7
    VSR[VRT+32].hword[i] ← EXT16(VSR[VRB+32].byte[i])
end

For each integer value i from 0 to 7, do the following.
The signed integer value in byte element i of VSR[VRB+32] is sign-extended and placed into halfword element i in VSR[VRT+32].

Special Registers Altered:
None

**Register Data Layout for vupkhs**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>96</td>
</tr>
</tbody>
</table>

**Vector Unpack Low Signed Byte VX-form**

vupklshb VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>654</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 7
    VSR[VRT+32].hword[i] ← EXT16(VSR[VRB+32].byte[i+8])
end

For each integer value i from 0 to 7, do the following.
The signed integer value in byte element i+8 of VSR[VRB+32] is sign-extended and placed into halfword element i of VSR[VRT+32].

Special Registers Altered:
None

**Register Data Layout for vupklsh**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
</tr>
</tbody>
</table>
Vector Unpack High Signed Halfword VX-form

\textbf{vupkhsh} \text{VRT,VRB}

\begin{tabular}{|c|c|c|c|}
\hline
4 & VRT & // & VRB \hline
0 & 6 & 11 & 16 & 21 & 31 \hline
\end{tabular}

\begin{itemize}
\item if MSR.VEC=0 then Vector_Unavailable()
\item VSR[VRT+32].word[0] ← EXTS32(VSR[VRB+32].hword[0])
\item VSR[VRT+32].word[1] ← EXTS32(VSR[VRB+32].hword[1])
\item VSR[VRT+32].word[2] ← EXTS32(VSR[VRB+32].hword[2])
\item VSR[VRT+32].word[3] ← EXTS32(VSR[VRB+32].hword[3])
\end{itemize}

For each integer value $i$ from 0 to 3, do the following.

The signed integer value in halfword element $i$ of VSR[VRB+32] is sign-extended and placed into word element $i$ in VSR[VRT+32].

Special Registers Altered:
None

\textbf{Register Data Layout for vupkhsh}

\begin{tabular}{|c|c|c|c|}
\hline
\end{tabular}

Vector Unpack Low Signed Halfword VX-form

\textbf{vupklsh} \text{VRT,VRB}

\begin{tabular}{|c|c|c|c|}
\hline
4 & VRT & // & VRB \hline
0 & 6 & 11 & 16 & 21 & 31 \hline
\end{tabular}

\begin{itemize}
\item if MSR.VEC=0 then Vector_Unavailable()
\item VSR[VRT+32].word[0] ← EXTS32(VSR[VRB+32].hword[4])
\item VSR[VRT+32].word[1] ← EXTS32(VSR[VRB+32].hword[5])
\item VSR[VRT+32].word[2] ← EXTS32(VSR[VRB+32].hword[6])
\item VSR[VRT+32].word[3] ← EXTS32(VSR[VRB+32].hword[7])
\end{itemize}

For each integer value $i$ from 0 to 3, do the following.

The signed integer value in halfword element $i+4$ of VSR[VRB+32] is sign-extended to produce a signed-integer word and placed into word element $i$ of VSR[VRT+32].

Special Registers Altered:
None

\textbf{Register Data Layout for vupklsh}

\begin{tabular}{|c|c|c|c|}
\hline
\end{tabular}
### Vector Unpack High Signed Word VX-form

**vupkhsw**

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1614</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[ \text{VSR[VRT+32].dword[0] ← EXT64(VSR[VRB+32].word[0])} \]
\[ \text{VSR[VRT+32].dword[1] ← EXT64(VSR[VRB+32].word[1])} \]

For each integer value \( i \) from 0 to 1, do the following.

The signed integer value in word element \( i \) of \( \text{VSR[VRB+32]} \) is sign-extended and placed into doubleword element \( i \) of \( \text{VSR[VRT+32]} \).

**Special Registers Altered:**
None

**Register Data Layout for vupkhsw**

<table>
<thead>
<tr>
<th>src 2</th>
<th>VSR[VRB+32].word[0]</th>
<th>VSR[VRB+32].word[1]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>48</td>
</tr>
</tbody>
</table>

### Vector Unpack Low Signed Word VX-form

**vupklsw**

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1742</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[ \text{VSR[VRT+32].dword[0] ← EXT64(VSR[VRB+32].word[2])} \]
\[ \text{VSR[VRT+32].dword[1] ← EXT64(VSR[VRB+32].word[3])} \]

For each integer value \( i \) from 0 to 1, do the following.

The signed integer value in word element \( i+2 \) of \( \text{VSR[VRB+32]} \) is sign-extended and placed into doubleword element \( i \) of \( \text{VSR[VRT+32]} \).

**Special Registers Altered:**
None

**Register Data Layout for vupklsw**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>48</td>
</tr>
</tbody>
</table>
Vector Unpack High Pixel VX-form

\[
\text{vupkhpx} \quad \text{VRT, VRB}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>846</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src ← VSR[VRB+32].hword[i]
    VSR[VRT+32].word[i].byte[0] ← EXT8(src.bit[0])
    VSR[VRT+32].word[i].byte[1] ← EXTZ(src.bit[1:5])
end

For each integer value \(i\) from 0 to 3, do the following.

The contents of halfword element \(i\) of VSR[VRB+32] are unpacked as follows.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1:5 of the halfword to 8 bits
- zero-extend bits 6:10 of the halfword to 8 bits
- zero-extend bits 11:15 of the halfword to 8 bits

The result is placed in word element \(i\) of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vupkhpx

\[
\begin{array}{c|cccccc}
\text{src} & \text{VSR[VRB+32].hword[0]} & \text{VSR[VRB+32].hword[1]} & \text{unused} & \text{VSR[VRB+32].hword[3]} \\
\hline
\text{result} & \text{VSR[VRT+32].word[0]} & \text{VSR[VRT+32].word[1]} & \text{VSR[VRT+32].word[2]} & \text{VSR[VRT+32].word[3]} \\
\end{array}
\]

Register Data Layout for vupklpx

\[
\begin{array}{c|cccccc}
\text{src} & \text{VSR[VRB+32].hword[4]} & \text{VSR[VRB+32].hword[5]} & \text{VSR[VRB+32].hword[6]} & \text{VSR[VRB+32].hword[7]} & \text{unused} \\
\hline
\text{result} & \text{VSR[VRT+32].word[0]} & \text{VSR[VRT+32].word[1]} & \text{VSR[VRT+32].word[2]} & \text{VSR[VRT+32].word[3]} \\
\end{array}
\]

Programming Note

The source and target elements can be considered to be 16-bit and 32-bit “pixels” respectively, having the formats described in the Programming Note for the Vector Pack Pixel instruction on page 278.

Programming Note

Notice that the unpacking done by the Vector Unpack Pixel instructions does not reverse the packing done by the Vector Pack Pixel instruction. Specifically, if a 16-bit pixel is unpacked to a 32-bit pixel which is then packed to a 16-bit pixel, the resulting 16-bit pixel will not, in general, be equal to the original 16-bit pixel (because, for each channel except the first, Vector Unpack Pixel inserts high-order bits while Vector Pack Pixel discards low-order bits).
### 6.8.3 Vector Merge Instructions

**Vector Merge High Byte VX-form**

vmrghb VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
  VSR[VRT+32].hword[i].byte[0] ← VSR[VRA+32].byte[i]  
end

For each integer value i from 0 to 7, do the following.
The contents of byte element i of VSR[VRA+32] are placed into byte element 2×i of VSR[VRT+32].
The contents of byte element i of VSR[VRB+32] are placed into byte element 2×i+1 of VSR[VRT+32].

Special Registers Altered:
None

**Register Data Layout for vmrghb**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
</tr>
</tbody>
</table>

**Vector Merge Low Byte VX-form**

vmrglb VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>268</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
  VSR[VRT+32].hword[i].byte[0] ← VSR[VRA+32].byte[i+8]  
end

For each integer value i from 0 to 7, do the following.
The contents of byte element i+8 of VSR[VRA+32] are placed into byte element 2×i of VSR[VRT+32].
The contents of byte element i+8 of VSR[VRB+32] are placed into byte element 2×i+1 of VSR[VRT+32].

Special Registers Altered:
None

**Register Data Layout for vmrglb**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
</tr>
</tbody>
</table>
**Vector Merge High Halfword VX-form**

vmrghh VRT, VRA, VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>76</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    VSR[VRT+32].word[i].hword[0] ← VSR[VRA+32].hword[i]
    VSR[VRT+32].word[i].hword[1] ← VSR[VRB+32].hword[i]
end

For each integer value i from 0 to 3, do the following.
The contents of halfword element i of VSR[VRA+32] are placed into halfword element 2xi of VSR[VRT+32].

The contents of halfword element i of VSR[VRB+32] are placed into halfword element 2xi+1 of VSR[VRT+32].

Special Registers Altered:
None

**Register Data Layout for vmrghh**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
<td>64</td>
</tr>
</tbody>
</table>

**Vector Merge Low Halfword VX-form**

vmrglh VRT, VRA, VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>332</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    VSR[VRT+32].word[i].hword[0] ← VSR[VRA+32].hword[i+4]
end

For each integer value i from 0 to 3, do the following.
The contents of halfword element i+4 of VSR[VRA+32] are placed into halfword element 2xi of VSR[VRT+32].

The contents of halfword element i+4 of VSR[VRB+32] are placed into halfword element 2xi+1 of VSR[VRT+32].

Special Registers Altered:
None

**Register Data Layout for vmrglh**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
<td>64</td>
</tr>
</tbody>
</table>
**Vector Merge High Word VX-form**

**vmrghw**  
VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>140</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

1. If MSR.VEC = 0 then Vector_Unavailable()

2. VSR[VRT+32].word[0] ← VSR[VRA+32].word[0]
3. VSR[VRT+32].word[1] ← VSR[VRB+32].word[0]

The contents of word element 0 of VSR[VRA+32] are placed into word element 0 of VSR[VRT+32].

The contents of word element 0 of VSR[VRB+32] are placed into word element 1 of VSR[VRT+32].

The contents of word element 1 of VSR[VRA+32] are placed into word element 2 of VSR[VRT+32].

The contents of word element 1 of VSR[VRB+32] are placed into word element 3 of VSR[VRT+32].

**Special Registers Altered:**

None

**Register Data Layout for vmrghw**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].word[0]</th>
<th>VSR[VRA+32].word[1]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRB+32].word[1]</td>
<td></td>
</tr>
</tbody>
</table>

| 0   | 32  | 64  | 48  | 127 |

**Vector Merge Low Word VX-form**

**vmrglw**  
VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>396</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

1. If MSR.VEC = 0 then Vector_Unavailable()

2. VSR[VRT+32].word[0] ← VSR[VRA+32].word[2]

The contents of word element 2 of VSR[VRA+32] are placed into word element 0 of VSR[VRT+32].

The contents of word element 2 of VSR[VRB+32] are placed into word element 1 of VSR[VRT+32].

The contents of word element 3 of VSR[VRA+32] are placed into word element 2 of VSR[VRT+32].

The contents of word element 3 of VSR[VRB+32] are placed into word element 3 of VSR[VRT+32].

**Special Registers Altered:**

None

**Register Data Layout for vmrglw**

|------|--------|---------------------|---------------------|

| 0   | 32  | 64  | 48  | 127 |
Vector Merge Even Word VX-form

```
vmrgew VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1932</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ← VSR[VRA+32].word[0]
VSR[VRT+32].word[1] ← VSR[VRA+32].word[0]

The contents of word element 0 of VSR[VRA+32] are placed into word element 0 of VSR[VRT+32].
The contents of word element 0 of VSR[VRB+32] are placed into word element 1 of VSR[VRT+32].
The contents of word element 2 of VSR[VRA+32] are placed into word element 2 of VSR[VRT+32].
The contents of word element 2 of VSR[VRB+32] are placed into word element 3 of VSR[VRT+32].

**vmrgew** is treated as a Vector instruction in terms of resource availability.

Special Registers Altered:
None

Register Data Layout for vmrgew

```
src1 VSR[VRA+32].word[0] unused VSR[VRA+32].word[2] unused
src2 VSR[VRB+32].word[0] unused VSR[VRB+32].word[2] unused
```

Vector Merge Odd Word VX-form

```
vmrgow VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1676</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ← VSR[VRA+32].word[1]

The contents of word element 1 of VSR[VRA+32] are placed into word element 0 of VSR[VRT+32].
The contents of word element 1 of VSR[VRB+32] are placed into word element 1 of VSR[VRT+32].
The contents of word element 3 of VSR[VRA+32] are placed into word element 2 of VSR[VRT+32].
The contents of word element 3 of VSR[VRB+32] are placed into word element 3 of VSR[VRT+32].

**vmrgow** is treated as a Vector instruction in terms of resource availability.

Special Registers Altered:
None

Register Data Layout for vmrgow

```
src1 unused VSR[VRA+32].word[1] unused VSR[VRA+32].word[3]
```
### 6.8.4 Vector Splat Instructions

#### Programming Note

The *Vector Splat* instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (e.g., multiplying all elements of a VSR by a constant).

#### Vector Splat Byte VX-form

**vspltb**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRB</th>
<th>UIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>524</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{if MSR.VEC}=0 \text{ then Vector.Unavailable()}
\]

\[
b \leftarrow \text{UIM} || 0b000
\]

\[
do \ i = 0 \text{ to } 15 \\
\ VSR[VRT+32].byte[i] \leftarrow VSR[VRB+32].bit[b:b+7]
\end{do}

For each integer value \(i\) from 0 to 15, do the following.

The contents of byte element \(UIM\) in \(VSR[VRB+32]\) are placed into byte element \(i\) of \(VSR[VRT+32]\).

**Special Registers Altered:** None

#### Vector Splat Halfword VX-form

**vsplth**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRB</th>
<th>UIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>588</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{if MSR.VEC}=0 \text{ then Vector.Unavailable()}
\]

\[
b \leftarrow \text{UIM} || 0b0000
\]

\[
do \ i = 0 \text{ to } 7 \\
\ VSR[VRT+32].hword[i] \leftarrow VSR[VRB+32].bit[b:b+15]
\end{do}

For each integer value \(i\) from 0 to 7, do the following.

The contents of halfword element \(UIM\) in \(VSR[VRB+32]\) are placed into halfword element \(i\) of \(VSR[VRT+32]\).

**Special Registers Altered:** None

#### Register Data Layout for vspltb

**Source**

\[
\begin{array}{cccccccccccc}
0 & 8 & 16 & 24 & 32 & 40 & 48 & 56 & 64 & 72 & 80 & 88 & 96 & 104 & 112 & 120 & 127
\end{array}
\]

**Result**

\[
\begin{array}{cccccccccccc}
0 & 8 & 16 & 24 & 32 & 40 & 48 & 56 & 64 & 72 & 80 & 88 & 96 & 104 & 112 & 120 & 127
\end{array}
\]

#### Register Data Layout for vsplth

**Source**

\[
\begin{array}{cccccccccccc}
0 & 8 & 16 & 24 & 32 & 40 & 48 & 56 & 64 & 72 & 80 & 88 & 96 & 104 & 112 & 120 & 127
\end{array}
\]

**Result**

\[
\begin{array}{cccccccccccc}
0 & 8 & 16 & 24 & 32 & 40 & 48 & 56 & 64 & 72 & 80 & 88 & 96 & 104 & 112 & 120 & 127
\end{array}
\]
Vector Splat Word VX-form

vspltw VRT,VRB,UIM

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>//</th>
<th>UW</th>
<th>VRB</th>
<th>652</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>14</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();

\[ b \leftarrow UIM || 000000 \]

do i = 0 to 3

\[ \text{VSR}[VRT+32].\text{word}[i] \leftarrow \text{VSR}[VRB+32].\text{bit}[b:b+31] \]

end

For each integer value \( i \) from 0 to 3, do the following.

The contents of word element \( UIM \) in \( \text{VSR}[VRB+32] \)
are placed into word element \( i \) of \( \text{VSR}[VRT+32] \).

Special Registers Altered:
None

Register Data Layout for vspltw

|-----|--------------------|--------------------|--------------------|--------------------|

0 32 64 48 127
**Vector Splat Immediate Signed Byte VX-form**

```plaintext
vspltisb VRT, SIM
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>SIM</th>
<th>///</th>
<th>780</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC = 0 then Vector_Unavailable()

Do i = 0 to 15

VSR[VRT+32].byte[i] ← EXT8(SIM, 8)

End

For each integer value i from 0 to 15, do the following.
The value of the SIM field, sign-extended to 8 bits,
is placed into byte element i of VSR[VRT+32].

**Special Registers Altered:** None

**Vector Splat Immediate Signed Halfword VX-form**

```plaintext
vspltish VRT, SIM
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>SIM</th>
<th>///</th>
<th>844</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC = 0 then Vector_Unavailable()

Do i = 0 to 7

VSR[VRT+32].hword[i] ← EXT16(SIM, 16)

End

For each integer value i from 0 to 7, do the following.
The value of the SIM field, sign-extended to 16 bits,
is placed into halfword element i of VSR[VRT+32].

**Special Registers Altered:** None

**Vector Splat Immediate Signed Word VX-form**

```plaintext
vspltisw VRT, SIM
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>SIM</th>
<th>///</th>
<th>908</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC = 0 then Vector_Unavailable()

Do i = 0 to 3

VSR[VRT+32].word[i] ← EXT32(SIM, 32)

End

For each integer value i from 0 to 3, do the following.
The value of the SIM field, sign-extended to 32 bits,
is placed into word element i of VSR[VRT+32].

**Special Registers Altered:** None

**Register Data Layout for vspltisb**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
<td>127</td>
</tr>
</tbody>
</table>

**Register Data Layout for vspltish**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
<td>64</td>
</tr>
<tr>
<td>64</td>
<td>80</td>
<td>96</td>
<td>112</td>
<td>127</td>
</tr>
</tbody>
</table>
6.8.5 Vector Permute Instruction

The Vector Permute instruction allows any byte in two source VSRs to be copied to any byte in the target VSR. The bytes in a third source VSR specify from which byte in the first two source VSRs the corresponding target byte is to be copied. The contents of the third source VSR are sometimes referred to as the “permute control vector”.

Vector Permute VA-form

\[
\begin{align*}
\text{vperm} & \quad \text{VRT, VRA, VRB, VRC} \\
\text{if MSR.VEC}=0 & \quad \text{then VectorUnavailable()} \\
vsrc.[0] & \quad \leftarrow \text{VSR[VRA+32]} \\
vsrc.[1] & \quad \leftarrow \text{VSR[VRB+32]} \\
\text{do } i = 0 \text{ to } 15 & \\
\text{index} & \quad \leftarrow \text{VSR[VRC+32].byte}[i].\text{bit}[3:7] \\
\text{VSR[VRT+32].byte}[i] & \quad \leftarrow \text{src.byte}[\text{index}] \\
\text{end}
\end{align*}
\]

Let the source vector be the concatenation of the contents of \text{VSR[VRA+32]} followed by the contents of \text{VSR[VRB+32]}.

For each integer value \(i\) from 0 to 15, do the following.

Let \text{index} be the value specified by bits 3:7 of byte element \(i\) of \text{VSR[VRC+32]}.

The contents of byte element \text{index} of \text{src} are placed into byte element \(i\) of \text{VSR[VRT+32]}.

Special Registers Altered: None

Programming Note

See the Programming Notes with the Load Vector for Shift Left and Load Vector for Shift Right instructions on page 276 for examples of uses of \text{vperm}.

Register Data Layout for vperm & vpermr

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{src1} & \text{src2} & \text{src3} & \text{result} \\
\hline
\text{VSR[VRA+32]} & \text{VSR[VRB+32]} & \text{VSR[VRC+32]} & \text{VSR[VRT+32]} \\
\hline
\end{array}
\]

Vector Permute Right-indexed VA-form

\[
\begin{align*}
\text{vpermr} & \quad \text{VRTX, VRA, VRB, VRC} \\
\text{if MSR.VEC}=0 & \quad \text{then VectorUnavailable()} \\
vsrc.[0] & \quad \leftarrow \text{VSR[VRA+32]} \\
vsrc.[1] & \quad \leftarrow \text{VSR[VRB+32]} \\
\text{do } i = 0 \text{ to } 15 & \\
\text{index} & \quad \leftarrow \text{VSR[VRC+32].byte}[i].\text{bit}[3:7] \\
\text{VSR[VRT+32].byte}[i] & \quad \leftarrow \text{src.byte}[31-\text{index}] \\
\text{end}
\end{align*}
\]

Let the source vector be the concatenation of the contents of \text{VSR[VRA+32]} followed by the contents of \text{VSR[VRT+32]}.

For each integer value \(i\) from 0 to 15, do the following.

Let \text{index} be the value specified by bits 3:7 of byte element \(i\) of \text{VSR[VRC+32]}.

The contents of byte element 31-index of \text{src} are placed into byte element \(i\) of \text{VSR[VRT+32]}.

Special Registers Altered: None
6.8.6 Vector Select Instruction

**Vector Select VA-form**

\[
\text{vsel} \quad \text{VRT, VRA, VRB, VRC}
\]

<table>
<thead>
<tr>
<th>3</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>VRC</td>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{if MSR.VEC=0 then Vector.Unavailable()}
\]

\[
src1 \leftarrow \text{VSR}[\text{VRA+32}]
\]

\[
src2 \leftarrow \text{VSR}[\text{VRB+32}]
\]

\[
\text{mask} \leftarrow \text{VSR}[\text{VRC+32}]
\]

\[
\text{VSR}[\text{VRT+32}] \leftarrow (\text{src1} \& \neg \text{mask}) \mid (\text{src2} \& \text{mask})
\]

Let \(src1\) be the contents of \(\text{VSR}[\text{VRA+32}]\).

Let \(src2\) be the contents of \(\text{VSR}[\text{VRB+32}]\).

Let \(mask\) be the contents of \(\text{VSR}[\text{VRC+32}]\).

The value, \((src1 \& \neg mask) \mid (src2 \& mask)\), is placed into \(\text{VSR}[\text{VRT+32}]\).

**Special Registers Altered:**

None

**Register Data Layout for vsel**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[VRC+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

0 127
6.8.7 Vector Shift Instructions

The Vector Shift instructions rotate or shift the contents of a VSR or a pair of VSRs left or right by a specified number of bytes (vslo, vsro, vsldoi) or bits (vsl, vsr). Depending on the instruction, this "shift count" is specified either by the contents of a VSR or by an immediate field in the instruction. In the former case, 7 bits of the shift count register give the shift count in bits ($0 \leq \text{count} \leq 127$). Of these 7 bits, the high-order 4 bits give the number of complete bytes by which to shift and are used by vslo and vsro; the low-order 3 bits give the number of remaining bits by which to shift and are used by vsl and vsr.

### Vector Shift Left Double by Bit Immediate VN-form

```plaintext
vsldbi VRT,VRA,VRB,SH
```

- **if MSR.VEC=0 then Vector_Unavailable()**
- vsrc.qword[0] ← VSR[VRA+32]
- vsrc.qword[1] ← VSR[VRB+32]
- VSR[VRT+32] ← vsrc.bit[SH:SH+127]

Let vsrc be the contents of VSR[VRA+32] concatenated with the contents of VSR[VRB+32].

The contents of bits SH:SH+127 of vsrc are placed into VSR[VRT+32].

**SH** can be any integer value between 0 and 7.

**Special Registers Altered:**

None

### Vector Shift Left Double by Octet Immediate VA-form

```plaintext
vsldoi VRT,VRA,VRB,SHB
```

- **if MSR.VEC=0 then Vector_Unavailable()**
- vsrc.qword[0] ← VSR[VRA+32]
- vsrc.qword[1] ← VSR[VRB+32]

Let vsrc be the contents of VSR[VRA+32] concatenated with the contents of VSR[VRB+32].

Bytes SHB:SHB+15 of vsrc are placed into VSR[VRT+32].

**Special Registers Altered:**

None

---

**Programming Note**

A pair of these instructions, specifying the same shift count register, can be used to shift the contents of a VSR left or right by the number of bits (0-127) specified in the shift count register. The following example shifts the contents of register Vx left by the number of bits specified in register Vy and places the result into register Vz.

vslo Vz, Vx, Vy
vspltb Vy, Vy, 15
vsl Vz, Vz, Vy
Vector Shift Right Double by Bit Immediate
VN-form

vsrdbi | VRT, VRA, VRB, SH

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

vsr.c.qword[0] ← VSR[VRA+32]
vsr.c.qword[1] ← VSR[VRB+32]


Let vsr.c be the contents of VSR[VRA+32] concatenated with the contents of VSR[VRB+32].

The contents of bits 128-SH:255-SH of vsr.c are placed into VSR[VRT+32].

SH can be any integer value between 0 and 7.

Special Registers Altered:
None

Register Data Layout for vsldbi, & vsrdbi & vsldoi

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

0 | 127 |
Vector Shift Left VX-form

\[
vsl VRT, VRA, VRB
\]

- if \( \text{MSR.VEC=0} \) then \( \text{Vector.Unavailable()} \)
- \( \text{shb} \leftarrow \text{VSR}[\text{VRB+32}].\text{bit}[125:127] \)
- \( \text{t} \leftarrow 1 \)
- \( \text{do } i = 0 \text{ to } 14 \)
  - \( \text{t} \leftarrow \text{t} \& (\text{VSR}[\text{VRB+32}].\text{byte}[i].\text{bit}[5:7] = \text{sh}) \)
- \( \text{end} \)
- if \( \text{t=1} \) then
  - \( \text{VSR}[\text{VRT+32}] \leftarrow \text{VSR}[\text{VRA+32}] \ll \text{sh} \)
- else
  - \( \text{VSR}[\text{VRT+32}] \leftarrow \text{UNDEFINED} \)

The contents of \( \text{VSR}[\text{VRA+32}] \) are shifted left by the number of bits specified in bits 125:127 of \( \text{VSR}[\text{VRB+32}] \).

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is place into \( \text{VSR}[\text{VRT+32}] \), except if, for any byte element in \( \text{VSR}[\text{VRB+32}] \), the low-order 3 bits are not equal to the shift amount, then \( \text{VSR}[\text{VRT+32}] \) is undefined.

Special Registers Altered:
None

Register Data Layout for \( \text{vsl} \& \text{vsr} \)

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRA+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

Vector Shift Right VX-form

\[
vsr VRT, VRA, VRB
\]

- if \( \text{MSR.VEC=0} \) then \( \text{Vector.Unavailable()} \)
- \( \text{sh} \leftarrow \text{VSR}[\text{VRB+32}].\text{bit}[125:127] \)
- \( \text{t} \leftarrow 1 \)
- \( \text{do } i = 0 \text{ to } 14 \)
  - \( \text{t} \leftarrow \text{t} \& (\text{VSR}[\text{VRB+32}].\text{byte}[i].\text{bit}[5:7] = \text{sh}) \)
- \( \text{end} \)
- if \( \text{t=1} \) then
  - \( \text{VSR}[\text{VRT+32}] \leftarrow \text{CHOP128(EXTZ(VSR[VRA+32]) >> sh)} \)
- else
  - \( \text{VSR}[\text{VRT+32}] \leftarrow \text{UNDEFINED} \)

The contents of \( \text{VSR}[\text{VRA+32}] \) are shifted right by the number of bits specified in bits 125:127 of \( \text{VSR}[\text{VRB+32}] \).

- Bits shifted out of bit 127 are lost.
- Zeros are supplied to the vacated bits on the left.

The result is place into \( \text{VSR}[\text{VRT+32}] \), except if, for any byte element in \( \text{VSR}[\text{VRB+32}] \), the low-order 3 bits are not equal to the shift amount, then \( \text{VSR}[\text{VRT+32}] \) is undefined.

Special Registers Altered:
None

Register Data Layout for \( \text{vsl} \& \text{vsr} \)

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRA+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
Vector Shift Left by Octet VX-form

```plaintext
vslo VRT, VRA, VRB
```

```
if MSR.VEC=0 then Vector_Unavailable()

shb ← VSR[VRB+32].bit[121:124] << 3
VSR[VRT+32] ← VSR[VRA+32] << shb
```

The contents of \( VSR[VRA+32] \) are shifted left by the number of bytes specified in bits 121:124 of \( VSR[VRB+32] \).

- Bytes shifted out of byte 0 are lost.
- Zeros are supplied to the vacated bytes on the right.

The result is placed into \( VSR[VRT+32] \).

**Special Registers Altered:**

- None

**Register Data Layout for vslo & vsro**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( VSR[VRA+32] )</td>
<td>( VSR[VRT+32] )</td>
<td>( VSR[VRT+32] )</td>
</tr>
</tbody>
</table>

**Programming Note**

A double-register shift by a dynamically specified number of bits (0-127) can be performed in six instructions. The following example shifts \( Vw \) left by the number of bits specified in \( Vy \) and places the high-order 128 bits of the result into \( Vz \).

```plaintext
vslo Vt1, Vw, Vy  # shift high-order reg left
vspltb Vy, Vy, 15
vs Vt1, Vt1, Vy
vsabnbm Vt3, V0, Vy  # adjust shift count (||VR||=0)
vsro Vt2, Vx, Vt3  # shift low-order reg right
vspltb Vt3, Vt3, 15
vs Vt2, Vt2, Vt3
vor Vz, Vt1, Vt2  # merge to get final result
```

Vector Shift Right by Octet VX-form

```plaintext
vsro VRT, VRA, VRB
```

```
if MSR.VEC=0 then Vector_Unavailable()

shb ← VSR[VRB+32].bit[121:124] << 3
VSR[VRT+32] ← VSR[VRA+32] >> shb
```

The contents of \( VSR[VRA+32] \) are shifted right by the number of bytes specified in bits 121:124 of \( VSR[VRB+32] \).

- Bytes shifted out of byte 15 are lost.
- Zeros are supplied to the vacated bytes on the left.

The result is placed into \( VSR[VRT+32] \).

**Special Registers Altered:**

- None
**Vector Shift Left Variable VX-form**

\[
\text{vslv} \quad \text{VRT}, \text{VRA}, \text{VRB}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1860</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable_Interrupt()

\[
\text{vsrc.byte}[0:15] \leftarrow \text{VSR}[\text{VRA}+32]
\]

\[
\text{vsrc.byte}[16] \leftarrow 0x00
\]

do i = 0 to 15

\[
\text{sh} \leftarrow \text{VSR}[\text{VRB}+32].\text{byte}[i].\text{bit}[5:7]
\]

\[
\text{VSR}[\text{VRT}+32].\text{byte}[i] \leftarrow \text{src.byte}[i:i+1].\text{bit}[\text{sh}:\text{sh}+7]
\]

end

Let bytes 0:15 of \text{vsrc} be the contents of \text{VSR}[\text{VRA}+32].

Let byte 16 of \text{vsrc} be the value 0x00.

For each integer value \(i\) from 0 to 15, do the following.

Let \(\text{sh}\) be the value in bits 5:7 of byte element \(i\) of \text{VSR}[\text{VRB}+32].

The contents of bits \(\text{sh}:\text{sh}+7\) of the halfword in byte elements \(i:i+1\) of \text{vsrc} are placed into byte element \(i\) of \text{VSR}[\text{VRT}+32].

**Register Data Layout for vslv & vsrv**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[0]</td>
<td>byte[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2</th>
<th>byte[0]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>byte[0]</th>
</tr>
</thead>
</table>

**Special Registers Altered:**

None

---

**Vector Shift Right Variable VX-form**

\[
\text{vsrv} \quad \text{VRT}, \text{VRA}, \text{VRB}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1796</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable_Interrupt()

\[
\text{vsrc.byte}[0] \leftarrow 0x00
\]

\[
\text{vsrc.byte}[1:16] \leftarrow \text{VSR}[\text{VRA}+32]
\]

do i = 0 to 15

\[
\text{sh} \leftarrow \text{VSR}[\text{VRB}+32].\text{byte}[i].\text{bit}[5:7]
\]

\[
\text{VSR}[\text{VRT}+32].\text{byte}[i] \leftarrow \text{src.byte}[i:i+1].\text{bit}[8-\text{sh}:15-\text{sh}]
\]

end

Let bytes 1:16 of \text{vsrc} be the contents of \text{VSR}[\text{VRA}+32].

Let byte 0 of \text{vsrc} be the value 0x00.

For each integer value \(i\) from 0 to 15, do the following.

Let \(\text{sh}\) be the value in bits 5:7 of byte element \(i\) of \text{VSR}[\text{VRB}+32].

The contents of bits \(8-\text{sh}:15-\text{sh}\) of the halfword in byte elements \(i:i+1\) of \text{vsrc} are placed into byte element \(i\) of \text{VSR}[\text{VRT}+32].

**Special Registers Altered:**

None
Chapter 6. Vector Facility

Version 3.1

Assume $vSRC$ contains a vector of packed 7-bit values, $A$ located in bits 0:6, $B$ located in bits 7:13, $C$ located in bits 14:20, etc..

```c
# vSRC = { 0bAAAAAAAB, 0bBBBBBBCC, 0bCCCCCDDD, 0bDDDDEEEE,
#          0bEEEFFFFF, 0bGGGGGGGG, 0bHHHHHHHH, 0bIIIIIIIIJ, 
#          0bJJJJJJKK, 0bKKKKKLLL, 0bLLLLMMMM, 0bMMMNNNNN, 
#          0bNNOOOOOO, 0bOPPPPPPP, 0bQQQQQQQR, 0bRRRRRRSS }; 
```

Assume the following registers are pre-loaded as follows,

```c
# vSHCNT1 = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x07, 
#             0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07 }; 
# vSHCNT2 = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 
#             0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x07, 0x07 }; 
# vSHCNT3 = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 
#             0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x02 }; 
# vMASK = { 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 
#          0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F }; 
```

The leftmost seven packed 7-bit values can be unpacked into byte elements 0 to 6 using $\text{vsrv}$ with $vSHCNT1$.

```c
vsrv vTMP1, vSRC, vSHCNT1
# vTMP1 = { 0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD, 
#          0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bGHHHHHHH, 
#          0bHIIIIIII, 0bIJJJJJJJ, 0bJKKKKKKK, 0bKLLLLLLL, 
#          0bLMMMMMMM, 0bMNNNNNNN, 0bNOOOOOOO, 0bPPPPPPPQ }; 
```

The next seven packed 7-bit values can then be unpacked into byte elements 7 to 13 using $\text{vsrv}$ with $vSHCNT2$.

```c
vsrv vTMP2, vTMP1, vSHCNT2
# vTMP2 = { 0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD, 
#          0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bGHHHHHHH, 
#          0bHIIIIIII, 0bIJJJJJJJ, 0bJKKKKKKK, 0bKLLLLLLL, 
#          0bLMMMMMMM, 0bMNNNNNNN, 0bNOOOOOOO, 0bPPPPPPPQ }; 
```

The next two packed 7-bit values can then be unpacked into byte elements 14 to 15 using $\text{vsrv}$ with $vSHCNT3$.

```c
vsrv vTMP3, vTMP2, vSHCNT3
# vTMP3 = { 0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD, 
#          0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bGHHHHHHH, 
#          0bHIIIIIII, 0bIJJJJJJJ, 0bJKKKKKKK, 0bKLLLLLLL, 
#          0bLMMMMMMM, 0bMNNNNNNN, 0bNOOOOOOO, 0bOPPPPPPQ }; 
```

The most-significant bit in each byte element is masked off to produce a vector of sixteen unsigned byte elements.

```c
vand vTMP4, vTMP3, vMASK
# vTMP4 = { 0b0AAAAAAA, 0b0BBBBBBB, 0b0CCCCCCC, 0b0DDDDDDD, 
#          0b0EEEEEEE, 0b0FFFFFFF, 0b0GGGGGGG, 0b0HHHHHHH, 
#          0b0IIIIIII, 0b0JJJJJJJ, 0b0KKKKKKK, 0b0LLLLLLL, 
#          0b0MNNNNNN, 0b0000000D, 0b0000000P }; 
```

The vector of sixteen unsigned byte elements can be further unpacked to two vectors of eight unsigned halfword elements using a $\text{vupkhsb}$ and a $\text{vupklsb}$.

```c
vupkhsb vTMP5, vTMP4
# vTMP5 = { 0b0000000D, 0bAAAAAAA, 0b0000000D, 0bBBBBBBB, ... }; 
vupklsb vTMP6, vTMP4
# vTMP6 = { 0b0000000D, 0bIIIIIII, 0b0000000D, 0bJJJJJJJ, ... }; 
```

The resultant two vectors of eight unsigned halfword elements can then be further unpacked to four vectors of four unsigned word elements using two $\text{vupkhsh}$ and two $\text{vupklsh}$ instructions.

```c
vupkhsh vRESULT0, vTMP5
# vRESULT0 = { 0b0000000D, 0b0000000D, 0b0000000D, 0bAAAAAAA, ... }; 
vupklsh vRESULT1, vTMP5
# vRESULT1 = { 0b0000000D, 0b0000000D, 0b0000000D, 0bBBBBBBB, ... }; 
vupkhsh vRESULT2, vTMP6
# vRESULT2 = { 0b0000000D, 0b0000000D, 0b0000000D, 0bIIIIIII, ... }; 
vupklsh vRESULT3, vTMP6
# vRESULT3 = { 0b0000000D, 0b0000000D, 0b0000000D, 0bJJJJJJJ, ... }; 
```
6.8.8 Vector Extract Element Instructions

6.8.8.1 Vector Extract Element to VSR using Immediate-specified Index Instructions

**Vector Extract Unsigned Byte to VSR using immediate-specified index VX-form**

\[ \text{vextractub} \ VRT, VRB, UIM \]

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{src} \leftarrow VSR[VRB+32].\text{byte}[UIM] \)
- \( VSR[VRT+32].\text{dword}[0] \leftarrow \text{EXTZ64} (\text{src}) \)
- \( VSR[VRT+32].\text{dword}[1] \leftarrow 0x0000_0000_0000_0000 \)

The contents of byte element \( UIM \) of \( VSR[VRB+32] \) are placed into bits 56:63 of \( VSR[VRT+32] \). The contents of the remaining byte elements of \( VSR[VRT+32] \) are set to 0.

**Special Registers Altered:**
- None

**Register Data Layout for vextractub**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( 63 )</td>
</tr>
<tr>
<td>( 0 \times 0000_0000_0000_0000 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>VSR[VRT+32].dword[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( 63 )</td>
</tr>
<tr>
<td>( 0 \times 0000_0000_0000_0000 )</td>
<td></td>
</tr>
</tbody>
</table>

**Vector Extract Unsigned Halfword to VSR using immediate-specified index VX-form**

\[ \text{vextractuh} \ VRT, VRB, UIM \]

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{src} \leftarrow VSR[VRB+32].\text{byte}[UIM:UIM+1] \)
- \( VSR[VRT+32].\text{dword}[0] \leftarrow \text{EXTZ64} (\text{src}) \)
- \( VSR[VRT+32].\text{dword}[1] \leftarrow 0x0000_0000_0000_0000 \)

The contents of byte elements \( UIM:UIM+1 \) of \( VSR[VRB+32] \) are placed into halfword element 3 of \( VSR[VRT+32] \). The contents of the remaining halfword elements of \( VSR[VRT+32] \) are set to 0.

- If the value of \( UIM \) is greater than 14, the results are undefined.

**Special Registers Altered:**
- None

**Register Data Layout for vextractuh**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( 63 )</td>
</tr>
<tr>
<td>( 0 \times 0000_0000_0000_0000 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>VSR[VRT+32].dword[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( 63 )</td>
</tr>
<tr>
<td>( 0 \times 0000_0000_0000_0000 )</td>
<td></td>
</tr>
</tbody>
</table>
Vector Extract Unsigned Word to VSR using immediate-specified index VX-form

\[vextractuw\] VRT, VRB, UIM

\[
\begin{array}{c|c|c|c|c}
4 & 6 & 11 & 16 & 21 \\
\hline
0 & 6 & 12 & 16 & 21 \\
\end{array}
\]

if MSR.VEC = 0 then Vector_Unavailable()

\[
src \leftarrow VSR[VRB+32].byte[UIM:UIM+3]
\]
\[
VSR[VRT+32].dword[0] \leftarrow \text{EXTZ}(src)
\]
\[
VSR[VRT+32].dword[1] \leftarrow 0x0000_0000_0000_0000
\]

The contents of byte elements \(UIM:UIM+3\) of \(VSR[VRB+32]\) are placed into word element 1 of \(VSR[VRT+32]\). The contents of the remaining word elements of \(VSR[VRT+32]\) are set to 0.

If the value of \(UIM\) is greater than 12, the results are undefined.

Special Registers Altered:
None

Register Data Layout for vextractuw

\[
\begin{array}{|c|c|c|c|c|}
\hline
src & VSR[VRB+32] \\
\hline
\end{array}
\]
\[
\begin{array}{|c|c|c|c|c|}
\hline
result & VSR[VRT+32].dword[0] & 0x0000_0000_0000_0000 \\
\hline
0 & 63 & 127 \\
\end{array}
\]

Vector Extract Doubleword to VSR using immediate-specified index VX-form

\[vextractd\] VRT, VRB, UIM

\[
\begin{array}{c|c|c|c|c}
4 & 6 & 11 & 16 & 21 \\
\hline
0 & 6 & 12 & 16 & 21 \\
\end{array}
\]

if MSR.VEC = 0 then Vector_Unavailable()

\[
src \leftarrow VSR[VRB+32].byte[UIM:UIM+7]
\]
\[
VSR[VRT+32].dword[0] \leftarrow src
\]
\[
VSR[VRT+32].dword[1] \leftarrow 0x0000_0000_0000_0000
\]

The contents of byte elements \(UIM:UIM+7\) of \(VSR[VRB+32]\) are placed into \(VSR[VRT+32]\). The contents of doubleword element 1 of \(VSR[VRT+32]\) are set to 0.

If the value of \(UIM\) is greater than 8, the results are undefined.

Special Registers Altered:
None

Register Data Layout for vextractd

\[
\begin{array}{|c|c|c|c|c|}
\hline
src & VSR[VRB+32] \\
\hline
\end{array}
\]
\[
\begin{array}{|c|c|c|c|c|}
\hline
result & VSR[VRT+32].dword[0] & 0x0000_0000_0000_0000 \\
\hline
0 & 63 & 127 \\
\end{array}
\]
6.8.8.2 Vector Extract Element to GPR using GPR-specified Index Instructions

**Vector Extract Unsigned Byte to GPR using GPR-specified Left-Index VX-form**

```
vextublx RT,RA,VRB
```

If MSR.VEC=0 then Vector_Unavailable() |

\[
\text{index} \leftarrow \text{GPR}[\text{RA}].\text{bit}[60:63]
\]

\[
\text{GPR}[\text{RT}] \leftarrow \text{EXTZ64}(\text{VSR}[\text{VRB+32}].\text{byte}[\text{index}])
\]

Let \text{index} be the contents of bits 60:63 of GPR[RA].

The contents of byte element \text{index} of VSR[VRB+32] are placed into bits 56:63 of GPR[RT].

The contents of bits 0:55 of GPR[RT] are set to 0.

**Special Registers Altered:**

None

**Register Data Layout for vextublx & vextubrx**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR[RA]</td>
<td>VSR[VRB+32]</td>
<td>GPR[RT]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1549</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>2</td>
<td>7</td>
<td>1805</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

**Vector Extract Unsigned Byte to GPR using GPR-specified Right-Index VX-form**

```
vextubrx RT,RA,VRB
```

If MSR.VEC=0 then Vector_Unavailable() |

\[
\text{index} \leftarrow \text{GPR}[\text{RA}].\text{bit}[60:63]
\]

\[
\text{GPR}[\text{RT}] \leftarrow \text{EXTZ64}(\text{VSR}[\text{VRB+32}].\text{byte}[15-\text{index}])
\]

Let \text{index} be the contents of bits 60:63 of GPR[RA].

The contents of byte element 15-\text{index} of VSR[VRB+32] are placed into bits 56:63 of GPR[RT].

The contents of bits 0:55 of GPR[RT] are set to 0.

**Special Registers Altered:**

None
Vector Extract Unsigned Halfword to GPR using GPR-specified Left-Index VX-form

\textbf{vextuhlx} \quad RT, RA, VRB

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 \\
\hline
0 & RT & RA & VRB & 1613 \\
\hline
\end{tabular}
\end{table}

if MSR.VEC=0 then \text{Vector_Unavailable}()

\text{index} \leftarrow \text{GPR[RA].bit[60:63]}
\text{GPR[RT]} \leftarrow \text{EXTZ64(VSR[VRB+32].byte[index:index+1])}

Let \text{index} be the contents of bits 60:63 of GPR[RA].

The contents of byte elements \text{index:index+1} of VSR[VRB+32] are placed into bits 48:63 of GPR[RT].

The contents of bits 0:47 of GPR[RT] are set to 0.

If the value of \text{index} is greater than 14, the results are undefined.

Special Registers Altered:
None

Register Data Layout for \text{vextuhlx} & \text{vextuhrx}

\begin{itemize}
\item \textbf{src1} \quad \text{GPR[RA]}
\item \textbf{src2} \quad \text{VSR[VRB+32]}
\item \textbf{result} \quad \text{GPR[RT]}
\end{itemize}

Vector Extract Unsigned Halfword to GPR using GPR-specified Right-Index VX-form

\textbf{vextuhrx} \quad RT, RA, VRB

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 \\
\hline
0 & RT & RA & VRB & 1869 \\
\hline
\end{tabular}
\end{table}

if MSR.VEC=0 then \text{Vector_Unavailable}()

\text{index} \leftarrow \text{GPR[RA].bit[60:63]}
\text{GPR[RT]} \leftarrow \text{EXTZ64(VSR[VRB+32].byte[14-index:15-index])}

Let \text{index} be the contents of bits 60:63 of GPR[RA].

The contents of byte elements 14-index:15-index of VSR[VRB+32] are placed into bits 48:63 of GPR[RT].

The contents of bits 0:47 of GPR[RT] are set to 0.

If the value of \text{index} is greater than 14, the results are undefined.

Special Registers Altered:
None
Vector Extract Unsigned Word to GPR using GPR-specified Left-Index VX-form

`vextuwlx RT,RA,VRB`

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{index} \leftarrow \text{GPR}[RA].\text{bit}[60:63] \)
- \( \text{GPR}[RT] \leftarrow \text{EXTZ64}(\text{VSR}[VRB+32].\text{byte}[\text{index}:\text{index}+3]) \)

Let \( \text{index} \) be the contents of bits 60:63 of \text{GPR}[RA].

The contents of byte elements \( \text{index}:\text{index}+3 \) of \text{VSR}[VRB+32] are placed into bits 32:63 of \text{GPR}[RT].

The contents of bits 0:31 of \text{GPR}[RT] are set to 0.

If the value of \( \text{index} \) is greater than 12, the results are undefined.

Special Registers Altered:
None

Register Data Layout for `vextuwlx` & `vextuwrx`

<table>
<thead>
<tr>
<th>src1</th>
<th>GPR[RA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>GPR[RT]</td>
</tr>
</tbody>
</table>

Vector ExtractUnsigned Word to GPR using GPR-specified Right-Index VX-form

`vextuwrx RT,RA,VRB`

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{index} \leftarrow \text{GPR}[RA].\text{bit}[60:63] \)
- \( \text{GPR}[RT] \leftarrow \text{EXTZ64}(\text{VSR}[VRB+32].\text{byte}[12-\text{index}:15-\text{index}]) \)

Let \( \text{index} \) be the contents of bits 60:63 of \text{GPR}[RA].

The contents of byte elements \( \text{index}:\text{index}+3 \) of \text{VSR}[VRB+32] are placed into bits 32:63 of \text{GPR}[RT].

The contents of bits 0:31 of \text{GPR}[RT] are set to 0.

If the value of \( \text{index} \) is greater than 12, the results are undefined.

Special Registers Altered:
None
### 6.8.8.3 Vector Extract Double Element to VSR Using GPR-specified Index Instructions

**Vector Extract Double Unsigned Byte to VSR using GPR-specified Left-Index VA-form**

**vextdubvlx**  
VRT, VRA, VRB, RC

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td></td>
<td></td>
<td>VRA</td>
<td></td>
<td>VB</td>
<td>RC</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RC].bit[59:63]

vsrc.qword[0] ← VSR[VRA+32]

vsrc.qword[1] ← VSR[VRB+32]

VSR[VRT+32].dword[0] ← EXTZ64(vsrc.byte[index])

VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000

Let index be the contents of bits 59:63 of GPR[RC].

Let vsrc be the concatenation of the contents of VSR[VRA+32] and VSR[VRB+32].

The contents of byte element index of vsrc are zero-extended and placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

**Special Registers Altered:**
None

**Register Data Layout for vextdubvlx & vextdubvrx**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>src3</td>
<td>GPR[RC]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>

**Vector Extract Double Unsigned Byte to VSR using GPR-specified Right-Index VA-form**

**vextdubvrx**  
VRT, VRA, VRB, RC

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td></td>
<td></td>
<td>VRA</td>
<td></td>
<td>VB</td>
<td>RC</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RC].bit[59:63]

vsrc.qword[0] ← VSR[VRA+32]

vsrc.qword[1] ← VSR[VRB+32]

VSR[VRT+32].dword[0] ← EXTZ64(vsrc.byte[31-index])

VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000

Let index be the contents of bits 59:63 of GPR[RC].

Let vsrc be the concatenation of the contents of VSR[VRA+32] and VSR[VRB+32].

The contents of byte element 31-index of vsrc are zero-extended and placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

**Special Registers Altered:**
None
Vector Extract Double Unsigned Halfword to VSR using GPR-specified Left-Index VA-form

\[ \text{vextduhvlx } \quad \text{VRT, VRA, VRB, RC} \]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 26 \\
0 & VRT & VRA & VRB & RC & 26 \\
\end{array}
\]

- If MSR.VEC=0 then Vector_Unavailable();
- \( \text{index} \leftarrow \text{GPR}[RC].\text{bit}[59:63] \)
- \( \text{vsrc}.\text{qword}[0] \leftarrow \text{VSR}[\text{VRA+32}] \)
- \( \text{vsrc}.\text{qword}[1] \leftarrow \text{VSR}[\text{VRB+32}] \)
- \( \text{VSR}[\text{VRT+32}].\text{dword}[0] \leftarrow \text{EXTZ64}(\text{vsrc}.\text{byte}[\text{index}:\text{index+1}]) \)
- \( \text{VSR}[\text{VRT+32}].\text{dword}[1] \leftarrow 0x0000_0000_0000_0000 \)

Let \( \text{index} \) be the contents of bits 59:63 of \( \text{GPR}[\text{RC}] \).

Let \( \text{vsrc} \) be the concatenation of the contents of \( \text{VSR}[\text{VRA+32}] \) and \( \text{VSR}[\text{VRB+32}] \).

The contents of byte elements \( \text{index}:\text{index+1} \) of \( \text{vsrc} \) are zero-extended and placed into doubleword 0 of \( \text{VSR}[\text{VRT+32}] \).

The contents of doubleword 1 of \( \text{VSR}[\text{VRT+32}] \) are set to 0.

If \( \text{index} \) is greater than 30, the result is undefined.

Special Registers Altered:
- None

Register Data Layout for \text{vextduhvlx} & \text{vextduhvrx}

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32]</td>
<td>VSR[VRB+32]</td>
<td>GPR[RC]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>

Vector Extract Double Unsigned Halfword to VSR using GPR-specified Right-Index VA-form

\[ \text{vextduhvrx } \quad \text{VRT, VRA, VRB, RC} \]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 27 \\
0 & VRT & VRA & VRB & RC & 27 \\
\end{array}
\]

- If MSR.VEC=0 then Vector_Unavailable();
- \( \text{index} \leftarrow \text{GPR}[\text{RC}].\text{bit}[59:63] \)
- \( \text{vsrc}.\text{qword}[0] \leftarrow \text{VSR}[\text{VRA+32}] \)
- \( \text{vsrc}.\text{qword}[1] \leftarrow \text{VSR}[\text{VRB+32}] \)
- \( \text{VSR}[\text{VRT+32}].\text{dword}[0] \leftarrow \text{EXTZ64}(\text{vsrc}.\text{byte}[30-\text{index}:31-\text{index}]) \)
- \( \text{VSR}[\text{VRT+32}].\text{dword}[1] \leftarrow 0x0000_0000_0000_0000 \)

Let \( \text{index} \) be the contents of bits 59:63 of \( \text{GPR}[\text{RC}] \).

Let \( \text{vsrc} \) be the concatenation of the contents of \( \text{VSR}[\text{VRA+32}] \) and \( \text{VSR}[\text{VRB+32}] \).

The contents of byte elements \( 30-\text{index}:31-\text{index} \) of \( \text{vsrc} \) are zero-extended and placed into doubleword 0 of \( \text{VSR}[\text{VRT+32}] \).

The contents of doubleword 1 of \( \text{VSR}[\text{VRT+32}] \) are set to 0.

If \( \text{index} \) is greater than 30, the result is undefined.

Special Registers Altered:
- None
Vector Extract Double Unsigned Word to VSR using GPR-specified Left-Index VA-form

vextduwlx VRT,VRA,VRB,RC

\[
\begin{array}{cccccc}
4 & 6 & VRT & 11 & VRA & 16 & VRB & 21 & RC & 26 & 28
\end{array}
\]

- if MSR.VEC=0 then Vector_Unavailable();
- index ← GPR[RC].bit[59:63]
- src.qword[0] ← VSR[VRA+32]
- src.qword[1] ← VSR[VRB+32]
- VSR[VRT+32].dword[0] ← EXTZ64(src.byte[index:index+3])
- VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000

Let index be the contents of bits 59:63 of GPR[RC].

Let src be the concatenation of the contents of VSR[VRA+32] and VSR[VRB+32].

The contents of byte elements index:index+3 of src are zero-extended and placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

If index is greater than 28, the result is undefined.

Special Registers Altered:
None

Register Data Layout for vextduwlx & vextduwr

src1 | VSR[VRA+32]
src2 | VSR[VRB+32]
src3 | GPR[RC]
result | VSR[VRT+32].dword[0] | 0x0000_0000_0000_0000

Vector Extract Double Unsigned Word to VSR using GPR-specified Right-Index VA-form

vextduwr VRT,VRA,VRB,RC

\[
\begin{array}{cccccc}
4 & 6 & VRT & 11 & VRA & 16 & VRB & 21 & RC & 26 & 29
\end{array}
\]

- if MSR.VEC=0 then Vector_Unavailable();
- index ← GPR[RC].bit[59:63]
- src.qword[0] ← VSR[VRA+32]
- src.qword[1] ← VSR[VRB+32]
- VSR[VRT+32].dword[0] ← EXTZ64(src.byte[28-index:index+3])
- VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000

Let index be the contents of bits 59:63 of GPR[RC].

Let src be the concatenation of the contents of VSR[VRA+32] and VSR[VRB+32].

The contents of byte elements 28-index:31-index of src are zero-extended and placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

If index is greater than 28, the result is undefined.

Special Registers Altered:
None
### Vector Extract Double Doubleword to VSR using GPR-specified Left-Index VA-form

**vextddvlx**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>RC</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()

- \( \text{index} \leftarrow \text{GPR[RC].bit}[59:63] \)
- \( \text{src.qword}[0] \leftarrow \text{VSR[VRA+32]} \)
- \( \text{src.qword}[1] \leftarrow \text{VSR[VRB+32]} \)

- \( \text{VSR[VRT+32].dword}[0] \leftarrow \text{src.byte}[\text{index:}\text{index}+7] \)
- \( \text{VSR[VRT+32].dword}[1] \leftarrow 0x0000_0000_0000_0000 \)

Let \( \text{index} \) be the contents of bits 59:63 of GPR[RC].

Let \( \text{src} \) be the concatenation of the contents of VSR[VRA+32] and VSR[VRB+32].

The contents of byte elements \( \text{index:}\text{index}+7 \) of \( \text{src} \) are placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

If \( \text{index} \) is greater than 24, the result is undefined.

**Special Registers Altered:**

None

### Vector Extract Double Doubleword to VSR using GPR-specified Right-Index VA-form

**vextddvrx**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>RC</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()

- \( \text{index} \leftarrow \text{GPR[RC].bit}[59:63] \)
- \( \text{src.qword}[0] \leftarrow \text{VSR[VRA+32]} \)
- \( \text{src.qword}[1] \leftarrow \text{VSR[VRB+32]} \)

- \( \text{VSR[VRT+32].dword}[0] \leftarrow \text{src.byte}[24-\text{index}:31-\text{index}] \)
- \( \text{VSR[VRT+32].dword}[1] \leftarrow 0x0000_0000_0000_0000 \)

Let \( \text{index} \) be the contents of bits 59:63 of GPR[RC].

Let \( \text{src} \) be the concatenation of the contents of VSR[VRA+32] and VSR[VRB+32].

The contents of byte elements \( 24-\text{index}:31-\text{index} \) of \( \text{src} \) are placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

If \( \text{index} \) is greater than 24, the result is undefined.

**Special Registers Altered:**

None

### Register Data Layout for vextddvlx & vextddvrx

- **src1**: VSR[VRA+32]
- **src2**: VSR[VRB+32]
- **src3**: GPR[RC]
- **result**: VSR[VRT+32].dword[0], 0x0000_0000_0000_0000
6.8.9 Vector Insert Element Instructions

6.8.9.1 Vector Insert Element from VSR Using Immediate-specified Index Instructions

**Vector Insert Byte from VSR using immediate-specified index VX-form**

```
vinsertb VRT,VRB,UIM
```

if MSR.VEC=0 then Vector_Unavailable()

\[ VSR[VRT+32].byte[UIM] \leftarrow VSR[VRB+32].byte[7] \]

The contents of byte element 7 of \( VSR[VRB+32] \) are placed into byte element \( UIM \) of \( VSR[VRT+32] \). The contents of the remaining byte elements of \( VSR[VRT+32] \) are not modified.

**Special Registers Altered:**
None

**Register Data Layout for vinsertb**

<table>
<thead>
<tr>
<th>src</th>
<th>unused</th>
<th>.byte[7]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>56</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>

**Vector Insert Halfword from VSR using immediate-specified index VX-form**

```
vinsrth VRT,VRB,UIM
```

if MSR.VEC=0 then Vector_Unavailable()


The contents of halfword element 3 of \( VSR[VRB+32] \) are placed into byte elements \( UIM:UIM+1 \) of \( VSR[VRT+32] \). The contents of the remaining byte elements of \( VSR[VRT+32] \) are not modified.

If the value of \( UIM \) is greater than 14, the results are undefined.

**Special Registers Altered:**
None

**Register Data Layout for vinsrth**

<table>
<thead>
<tr>
<th>src</th>
<th>unused</th>
<th>VSR[VRB+32].hword[1]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>64</td>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>
### Vector Insert Word from VSR using immediate-specified index VX-form

**vinsertw** VRT,VRB,UIM

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>909</th>
</tr>
</thead>
</table>

**Constraints**
- If MSR.VEC=0 then Vector_Unavailable()

**Operation**

\[
\text{VSR[VRT+32].byte[UIM:UIM+3]} \leftarrow \text{VSR[VRB+32].word[1]}
\]

The contents of word element 1 of VSR[VRB+32] are placed into byte elements UIM:UIM+3 of VSR[VRT+32]. The contents of the remaining byte elements of VSR[VRT+32] are not modified.

If the value of UIM is greater than 12, the results are undefined.

**Special Registers Altered:** None

**Register Data Layout for vinsertw**

<table>
<thead>
<tr>
<th>src</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>VSR[VRT+32].word[1]</td>
</tr>
<tr>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

### Vector Insert Doubleword from VSR using immediate-specified index VX-form

**vinsertd** VRT,VRB,UIM

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>973</th>
</tr>
</thead>
</table>

**Constraints**
- If MSR.VEC=0 then Vector_Unavailable()

**Operation**

\[
\text{VSR[VRT+32].byte[UIM:UIM+7]} \leftarrow \text{VSR[VRB+32].dword[0]}
\]

The contents of doubleword element 0 of VSR[VRB+32] are placed into byte elements UIM:UIM+7 of VSR[VRT+32]. The contents of the remaining byte elements of VSR[VRT+32] are not modified.

If the value of UIM is greater than 8, the results are undefined.

**Special Registers Altered:** None

**Register Data Layout for vinsertd**

<table>
<thead>
<tr>
<th>src</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>
6.8.9.2 Vector Insert Element from GPR Using GPR-specified Index Instructions

**Vector Insert Byte from GPR using GPR-specified Left-Index VX-form**

```assembly
vinsblx VRT,RA,RB
```

```assembly
if MSR.VEC=0 then Vector_Unavailable();

index ← GPR[RA].bit[60:63]
src.byte[0:15] ← 0
VSR[VRT+32].byte[index] ← GPR[RB].bit[56:63]
```

Let \( index \) be the contents of bits 60:63 of GPR[RA].

The contents of bits 56:63 of GPR[RB] are placed into byte element \( index \) of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

**Special Registers Altered:**

None

---

**Vector Insert Byte from GPR using GPR-specified Right-Index VX-form**

```assembly
vinsbrx VRT,RA,RB
```

```assembly
if MSR.VEC=0 then Vector_Unavailable();

index ← GPR[RA].bit[60:63]
src.byte[0:15] ← 0
```

Let \( index \) be the contents of bits 60:63 of GPR[RA].

The contents of bits 56:63 of GPR[RB] are placed into byte element 15-\( index \) of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

**Special Registers Altered:**

None

---

**Register Data Layout for vinshlx & vinshrx**

<table>
<thead>
<tr>
<th>src1</th>
<th>GPR[RA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>unused</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

0 56 64 127
Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form

```
vinslx VRT,RA,RB
<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>RA</th>
<th>RB</th>
<th>591</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable() index ← GPR[RA].bit[60:63]

Let index be the contents of bits 60:63 of GPR[RA]. The contents of bits 48:63 of GPR[RB] are placed into byte elements index:index+1 of VSR[VRT+32].

If index is greater than 14, the result is undefined.

Special Registers Altered: None
```

Vector Insert Halfword from GPR using GPR-specified Right-Index VX-form

```
vinsrx VRT,RA,RB
<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>RA</th>
<th>RB</th>
<th>847</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable() index ← GPR[RA].bit[60:63]

Let index be the contents of bits 60:63 of GPR[RA]. The contents of bits 48:63 of GPR[RB] are placed into byte elements 14-index:15-index of VSR[VRT+32].

If index is greater than 14, the result is undefined.

Special Registers Altered: None
```

Register Data Layout for vinslx & vinsrx

```
src1                  GPR[RA]
src2                  unused        GPR[RB].hword[3]
result                VSR[VRT+32]
```

Version 3.1
**Vector Insert Word from GPR using GPR-specified Left-Index VX-form**

vinswlx VRT, RA, RB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>RA</th>
<th>RB</th>
<th>655</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]


Let index be the contents of bits 60:63 of GPR[RA].

The contents of bits 32:63 of GPR[RB] are placed into byte elements index:index+3 of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

If index is greater than 12, the result is undefined.

**Special Registers Altered:**

None

---

**Vector Insert Word from GPR using GPR-specified Right-Index VX-form**

vinswrx VRT, RA, RB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>RA</th>
<th>RB</th>
<th>911</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]


Let index be the contents of bits 60:63 of GPR[RA].

The contents of bits 32:63 of GPR[RB] are placed into byte elements 12-index:15-index of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

If index is greater than 12, the result is undefined.

**Special Registers Altered:**

None

---

**Register Data Layout for vinswlx & vinswrx**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
</table>

0 32 64 127
Vector Insert Doubleword from GPR using GPR-specified Left-Index VX-form

\( \text{vinsdlx} \ VRT, RA, RB \)

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>RA</th>
<th>RB</th>
<th>719</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

VSR[VRT+32].byte[index:index+7] ← GPR[RB]

Let index be the contents of bits 60:63 of GPR[RA].

The contents of GPR[RB] are placed into byte elements index:index+7 of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

If index is greater than 8, the result is undefined.

Special Registers Altered:
None

Register Data Layout for vinsdlx & vinsdrx

\[\begin{array}{c}
\text{src1} \\
\text{src2} \\
\text{result}
\end{array} \]

\[\begin{array}{cccc}
\text{GPR[RA]} \\
\text{GPR[RB]} \\
\text{VSR[VRT+32]}
\end{array} \]

Vector Insert Doubleword from GPR using GPR-specified Right-Index VX-form

\( \text{vinsdrx} \ VRT, RA, RB \)

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>RA</th>
<th>RB</th>
<th>975</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

VSR[VRT+32].byte[8-index:15-index] ← GPR[RB]

Let index be the contents of bits 60:63 of GPR[RA].

The contents of GPR[RB] are placed into byte elements 8-index:15-index of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

If index is greater than 8, the result is undefined.

Special Registers Altered:
None

Register Data Layout for vinsdlx & vinsdrx

\[\begin{array}{c}
\text{src1} \\
\text{src2} \\
\text{result}
\end{array} \]

\[\begin{array}{cccc}
\text{GPR[RA]} \\
\text{GPR[RB]} \\
\text{VSR[VRT+32]}
\end{array} \]
Vector Insert Word from GPR using immediate-specified index VX-form

\[ \text{vinsw} \ VRT, RB, UIM \]

- If MSR.VEC = 0 then Vector_Unavailable()
- \[ \text{VSR}[VRT+32].\text{byte}[UIM:UIM+3] \leftarrow \text{GPR}[RB].\text{bit}[32:63] \]

The contents of bits 32:63 of GPR[RB] are placed into byte elements UIM:UIM+3 of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

If UIM is greater than 12, the result is undefined.

Special Registers Altered:
None

Register Data Layout for vinsw

<table>
<thead>
<tr>
<th>src</th>
<th>unused</th>
<th>GPR(RB).word[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

Vector Insert Doubleword from GPR using immediate-specified index VX-form

\[ \text{vinsd} \ VRT, RB, UIM \]

- If MSR.VEC = 0 then Vector_Unavailable()
- \[ \text{VSR}[VRT+32].\text{byte}[UIM:UIM+7] \leftarrow \text{GPR}[RB] \]

The contents of GPR[RB] are placed into byte elements UIM:UIM+7 of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

If UIM is greater than 8, the result is undefined.

Special Registers Altered:
None

Register Data Layout for vinsd

<table>
<thead>
<tr>
<th>src</th>
<th>GPR(RB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>
### 6.8.9.4 Vector Insert Element from VSR Using GPR-specified Index Instructions

#### Vector Insert Byte from VSR using GPR-specified Left-Index VX-form

\[ \text{vinsbvlx} \quad \text{VRT}, \text{RA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>15</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VRT</td>
<td>RA</td>
<td>VRB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{index} \leftarrow \text{GPR[RA].bit}[60:63] \)
- \( \text{VSR}[\text{VRTX+32}].\text{byte}[\text{index}] \leftarrow \text{VSR}[\text{VRB+32}].\text{bit}[56:63] \)

Let \( \text{index} \) be the contents of bits 60:63 of \( \text{GPR[RA]} \).

The contents of bits 56:63 of \( \text{VSR}[\text{VRB+32}] \) are placed into byte element \( \text{index} \) of \( \text{VSR}[\text{VRTX+32}] \).

All other byte elements of \( \text{VSR}[\text{VRTX+32}] \) are not modified.

**Special Registers Altered:**
None

#### Register Data Layout for vinsbvlx & vinsbvr

<table>
<thead>
<tr>
<th>src1</th>
<th>GPR[RA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>unused</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>0</td>
<td>56</td>
</tr>
</tbody>
</table>

#### Vector Insert Byte from VSR using GPR-specified Right-Index VX-form

\[ \text{vinsbvrx} \quad \text{VRT}, \text{RA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>271</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VRT</td>
<td>RA</td>
<td>VRB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{index} \leftarrow \text{GPR[RA].bit}[60:63] \)
- \( \text{VSR}[\text{VRTX+32}].\text{byte}[15-\text{index}] \leftarrow \text{VSR}[\text{VRB+32}].\text{bit}[56:63] \)

Let \( \text{index} \) be the contents of bits 60:63 of \( \text{GPR[RA]} \).

The contents of bits 56:63 of \( \text{VSR}[\text{VRB+32}] \) are placed into byte element \( 15-\text{index} \) of \( \text{VSR}[\text{VRTX+32}] \).

All other byte elements of \( \text{VSR}[\text{VRTX+32}] \) are not modified.

**Special Registers Altered:**
None
### Vector Insert Halfword from VSR using GPR-specified Left-Index VX-form

**vinshvlx**  
**VRT,RA,VRB**

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>79</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

- \( \text{if MSR.VEC}=0 \) then Vector_Unavailable()
- \( \text{index} \leftarrow \text{GPR}[RA].\text{bit}[60:63] \)
- \( \text{src.byte}[0:15] \leftarrow 0 \)
- \( \text{VSR}[VRT+32].\text{byte}[\text{index}:%\text{index}+1] \leftarrow \text{VSR}[VRB+32].\text{bit}[48:63] \)

Let \( \text{index} \) be the contents of bits 60:63 of GPR[RA].

The contents of bits 48:63 of VSR[VRB+32] are placed into byte elements \( \text{index}:%\text{index}+1 \) of VSR[VRT+32].

If \( \text{index} \) is greater than 14, the result is undefined.

**Special Registers Altered:**

None

### Register Data Layout for vinshvlx & vinshvrx

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPR[RA]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unused</td>
<td>[VRB+32].hword[1]</td>
</tr>
<tr>
<td>0</td>
<td>48</td>
<td>64</td>
</tr>
</tbody>
</table>

### Vector Insert Halfword from VSR using GPR-specified Right-Index VX-form

**vinshvrx**  
**VRT,RA,VRB**

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>335</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

- \( \text{if MSR.VEC}=0 \) then Vector_Unavailable()
- \( \text{index} \leftarrow \text{GPR}[RA].\text{bit}[60:63] \)
- \( \text{src.byte}[0:15] \leftarrow 0 \)
- \( \text{VSR}[VRT+32].\text{byte}[14-%\text{index}:15-%\text{index}] \leftarrow \text{VSR}[VRB+32].\text{bit}[48:63] \)

Let \( \text{index} \) be the contents of bits 60:63 of GPR[RA].

The contents of bits 48:63 of VSR[VRB+32] are placed into byte elements \( 14-%\text{index}:15-%\text{index} \) of VSR[VRT+32].

If \( \text{index} \) is greater than 14, the result is undefined.

**Special Registers Altered:**

None
**Vector Insert Word from VSR using GPR-specified Left-Index VX-form**

\[
\text{vinswvlx} \quad \text{VRT, RA, VRB}
\]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

\[
\text{if MSR.VEC}=0 \text{ then Vector_Unavailable}()
\]

\[
\text{index} \leftarrow \text{GPR}[\text{RA}].\text{bit}[60:63]
\]

\[
\text{VSR}[\text{VRT+32}].\text{byte}[\text{index}:\text{index}+3] \leftarrow \text{VSR}[\text{VRB+32}].\text{bit}[32:63]
\]

Let \text{index} be the contents of bits 60:63 of GPR[RA].

The contents of bits 32:63 of VSR[VRB+32] are placed into byte elements \text{index}:\text{index}+3 of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

If \text{index} is greater than 12, the result is undefined.

**Special Registers Altered:**

None

**Register Data Layout for vinswvlx & vinswvrx**

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{src1} & \text{GPR( RA )} & \text{src2} & \text{unused} & \text{VSR}[ \text{VRB+32}].\text{word}[1] & \text{unused} \\
\hline
\text{result} & \text{VSR}[ \text{VRT+32}] & & & & \\
0 & 32 & 64 & 127 & & \\
\hline
\end{array}
\]

**Vector Insert Word from VSR using GPR-specified Right-Index VX-form**

\[
\text{vinswvrx} \quad \text{VRT, RA, VRB}
\]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

\[
\text{if MSR.VEC}=0 \text{ then Vector_Unavailable}()
\]

\[
\text{index} \leftarrow \text{GPR}[\text{RA}].\text{bit}[60:63]
\]

\[
\text{VSR}[\text{VRT+32}].\text{byte}[12-\text{index}:15-\text{index}] \leftarrow \text{VSR}[\text{VRB+32}].\text{bit}[32:63]
\]

Let \text{index} be the contents of bits 60:63 of GPR[RA].

The contents of bits 32:63 of VSR[VRB+32] are placed into byte elements 12-\text{index}:15-\text{index} of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

If \text{index} is greater than 12, the result is undefined.

**Special Registers Altered:**

None
6.9 Vector Integer Instructions

6.9.1 Vector Integer Arithmetic Instructions

6.9.1.1 Vector Integer Add Instructions

**Vector Add & Write Carry-out Unsigned Word VX-form**

vaddcuw VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>384</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3

src1 ← EXTZ(VSR[VRA+32].word[i])

src2 ← EXTZ(VSR[VRB+32].word[i])

VSR[VRT+32].word[i] ← CHOP32((src1 + src2) >> 32)

do end

For each integer value i from 0 to 3, do the following.

The unsigned integer value in word element i in VSR[VRA+32] is added to the unsigned integer value in word element i in VSR[VRB+32]. The carry out of the 32-bit sum is zero-extended to 32 bits and placed into word element i of VSR[VRT+32].

**Special Registers Altered:**

None

**Register Data Layout for vaddcuw**

|------|---------------------|---------------------|---------------------|---------------------|

| 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 | 104 | 112 | 120 | 127 |

**Vector Add Signed Byte Saturate VX-form**

vaddsbs VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>768</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

-do i = 0 to 15

src1 ← EXT5(VSR[VRA+32].byte[i])

src2 ← EXT5(VSR[VRB+32].byte[i])

VSR[VRT+32].byte[i] ← si8_CLAMP(src1 + src2)

do end

For each integer value i from 0 to 15, do the following.

The signed integer value in byte element i of VSR[VRA+32] is added to the signed integer value in byte element i of VSR[VRB+32].

- If the sum is greater than $2^{7} - 1$ the result saturates to $2^{7} - 1$ and SAT is set to 1.
- If the sum is less than $-2^{7}$ the result saturates to $-2^{7}$ and SAT is set to 1.

The result is placed into byte element i of VSR[VRT+32].

**Special Registers Altered:**

SAT

**Register Data Layout for vaddsbs**

|------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|

| 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 | 104 | 112 | 120 | 127 |
Vector Add Signed Halfword Saturate VX-form

The signed integer value in halfword element \(i\) of VSR\([VRA+32]\) is added to the signed integer value in halfword element \(i\) of VSR\([VRB+32]\).

- If the sum is greater than \(2^{15} - 1\) the result saturates to \(2^{15} - 1\) and SAT is set to 1.
- If the sum is less than \(-2^{15}\) the result saturates to \(-2^{15}\) and SAT is set to 1.

The result is placed into halfword element \(i\) of VSR\([VRT+32]\).

Special Registers Altered:

- SAT

Register Data Layout for vaddshs

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR([VRA+32].)hword(0)</td>
<td>VSR([VRA+32].)hword(1)</td>
<td>VSR([VRT+32].)hword(0)</td>
</tr>
<tr>
<td>VSR([VRA+32].)hword(2)</td>
<td>VSR([VRA+32].)hword(3)</td>
<td>VSR([VRT+32].)hword(1)</td>
</tr>
<tr>
<td>VSR([VRA+32].)hword(4)</td>
<td>VSR([VRA+32].)hword(5)</td>
<td>VSR([VRT+32].)hword(2)</td>
</tr>
<tr>
<td>VSR([VRA+32].)hword(6)</td>
<td>VSR([VRA+32].)hword(7)</td>
<td>VSR([VRT+32].)hword(3)</td>
</tr>
<tr>
<td>VSR([VRB+32].)hword(0)</td>
<td>VSR([VRB+32].)hword(1)</td>
<td>VSR([VRT+32].)hword(4)</td>
</tr>
<tr>
<td>VSR([VRB+32].)hword(2)</td>
<td>VSR([VRB+32].)hword(3)</td>
<td>VSR([VRT+32].)hword(5)</td>
</tr>
<tr>
<td>VSR([VRB+32].)hword(4)</td>
<td>VSR([VRB+32].)hword(5)</td>
<td>VSR([VRT+32].)hword(6)</td>
</tr>
<tr>
<td>VSR([VRB+32].)hword(6)</td>
<td>VSR([VRB+32].)hword(7)</td>
<td>VSR([VRT+32].)hword(7)</td>
</tr>
</tbody>
</table>

Vector Add Signed Word Saturate VX-form

The signed integer value in word element \(i\) of VSR\([VRA+32]\) is added to the signed integer value in word element \(i\) of VSR\([VRB+32]\).

- If the sum is greater than \(2^{31} - 1\) the result saturates to \(2^{31} - 1\) and SAT is set to 1.
- If the sum is less than \(-2^{31}\) the result saturates to \(-2^{31}\) and SAT is set to 1.

The result is placed into word element \(i\) of VSR\([VRT+32]\).

Special Registers Altered:

- SAT

Register Data Layout for vaddsws

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR([VRA+32].)word(0)</td>
<td>VSR([VRA+32].)word(1)</td>
<td>VSR([VRT+32].)word(0)</td>
</tr>
<tr>
<td>VSR([VRA+32].)word(2)</td>
<td>VSR([VRA+32].)word(3)</td>
<td>VSR([VRT+32].)word(1)</td>
</tr>
<tr>
<td>VSR([VRA+32].)word(4)</td>
<td>VSR([VRA+32].)word(5)</td>
<td>VSR([VRT+32].)word(2)</td>
</tr>
<tr>
<td>VSR([VRA+32].)word(6)</td>
<td>VSR([VRA+32].)word(7)</td>
<td>VSR([VRT+32].)word(3)</td>
</tr>
<tr>
<td>VSR([VRB+32].)word(0)</td>
<td>VSR([VRB+32].)word(1)</td>
<td>VSR([VRT+32].)word(4)</td>
</tr>
<tr>
<td>VSR([VRB+32].)word(2)</td>
<td>VSR([VRB+32].)word(3)</td>
<td>VSR([VRT+32].)word(5)</td>
</tr>
<tr>
<td>VSR([VRB+32].)word(4)</td>
<td>VSR([VRB+32].)word(5)</td>
<td>VSR([VRT+32].)word(6)</td>
</tr>
<tr>
<td>VSR([VRB+32].)word(6)</td>
<td>VSR([VRB+32].)word(7)</td>
<td>VSR([VRT+32].)word(7)</td>
</tr>
</tbody>
</table>
Vector Add Unsigned Byte Modulo VX-form

\textbf{vaddubm} \ VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>0</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable();

\begin{align*}
\text{do } i &= 0 \text{ to } 15 \\
\text{src1} &\leftarrow \text{EXTZ(VSR}[\text{VRA+32}].\text{byte}[i]) \\
\text{src2} &\leftarrow \text{EXTZ(VSR}[\text{VRB+32}].\text{byte}[i]) \\
\text{VSR}[\text{VRT+32}].\text{byte}[i] &\leftarrow \text{CHOP8(src1 + src2)}
\end{align*}

end

For each integer value \( i \) from 0 to 15, do the following.

The integer value in byte element \( i \) of VSR[VRT+32] is added to the integer value in byte element \( i \) of VSR[VRB+32].

The low-order 8 bits of the result are placed into byte element \( i \) of VSR[VRT+32].

Special Registers Altered:

None

\textbf{Programming Note}

\texttt{vaddubm} can be used for unsigned or signed-integers.

Register Data Layout for \texttt{vaddubm}

\begin{align*}
\text{src1} &\colon \text{byte}[0] \quad \text{byte}[1] \\
&\quad \text{byte}[2] \quad \text{byte}[3] \\
&\quad \text{byte}[4] \quad \text{byte}[5] \\
&\quad \text{byte}[6] \quad \text{byte}[7] \\
&\quad \text{byte}[8] \quad \text{byte}[9] \\
&\quad \text{byte}[10] \quad \text{byte}[11] \\
&\quad \text{byte}[12] \quad \text{byte}[13] \\
&\quad \text{byte}[14] \quad \text{byte}[15] \\
\text{src2} &\colon \text{byte}[0] \quad \text{byte}[1] \\
&\quad \text{byte}[2] \quad \text{byte}[3] \\
&\quad \text{byte}[4] \quad \text{byte}[5] \\
&\quad \text{byte}[6] \quad \text{byte}[7] \\
&\quad \text{byte}[8] \quad \text{byte}[9] \\
&\quad \text{byte}[10] \quad \text{byte}[11] \\
&\quad \text{byte}[12] \quad \text{byte}[13] \\
&\quad \text{byte}[14] \quad \text{byte}[15] \\
\text{result} &\colon \text{byte}[0] \quad \text{byte}[1] \\
&\quad \text{byte}[2] \quad \text{byte}[3] \\
&\quad \text{byte}[4] \quad \text{byte}[5] \\
&\quad \text{byte}[6] \quad \text{byte}[7] \\
&\quad \text{byte}[8] \quad \text{byte}[9] \\
&\quad \text{byte}[10] \quad \text{byte}[11] \\
&\quad \text{byte}[12] \quad \text{byte}[13] \\
&\quad \text{byte}[14] \quad \text{byte}[15]
\end{align*}

Vector Add Unsigned Halfword Modulo VX-form

\textbf{vadduhm} \ VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>64</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable();

\begin{align*}
\text{do } i &= 0 \text{ to } 7 \\
\text{src1} &\leftarrow \text{EXTZ(VSR}[\text{VRA+32}].\text{hword}[i]) \\
\text{src2} &\leftarrow \text{EXTZ(VSR}[\text{VRB+32}].\text{hword}[i]) \\
\text{VSR}[\text{VRT+32}].\text{hword}[i] &\leftarrow \text{CHOP16(src1 + src2)}
\end{align*}

end

For each integer value \( i \) from 0 to 7, do the following.

The integer value in halfword element \( i \) of VSR[VRT+32] is added to the integer value in halfword element \( i \) of VSR[VRB+32].

The low-order 16 bits of the result are placed into halfword element \( i \) of VSR[VRT+32].

Special Registers Altered:

None

\textbf{Programming Note}

\texttt{vadduhm} can be used for unsigned or signed-integers.

Register Data Layout for \texttt{vadduhm}

\begin{align*}
\text{src1} &\colon \text{hword}[0] \quad \text{hword}[1] \\
&\quad \text{hword}[2] \quad \text{hword}[3] \\
&\quad \text{hword}[4] \quad \text{hword}[5] \\
&\quad \text{hword}[6] \quad \text{hword}[7] \\
&\quad \text{hword}[8] \quad \text{hword}[9] \\
&\quad \text{hword}[10] \quad \text{hword}[11] \\
&\quad \text{hword}[12] \quad \text{hword}[13] \\
&\quad \text{hword}[14] \quad \text{hword}[15] \\
\text{src2} &\colon \text{hword}[0] \quad \text{hword}[1] \\
&\quad \text{hword}[2] \quad \text{hword}[3] \\
&\quad \text{hword}[4] \quad \text{hword}[5] \\
&\quad \text{hword}[6] \quad \text{hword}[7] \\
&\quad \text{hword}[8] \quad \text{hword}[9] \\
&\quad \text{hword}[10] \quad \text{hword}[11] \\
&\quad \text{hword}[12] \quad \text{hword}[13] \\
&\quad \text{hword}[14] \quad \text{hword}[15] \\
\text{result} &\colon \text{hword}[0] \quad \text{hword}[1] \\
&\quad \text{hword}[2] \quad \text{hword}[3] \\
&\quad \text{hword}[4] \quad \text{hword}[5] \\
&\quad \text{hword}[6] \quad \text{hword}[7] \\
&\quad \text{hword}[8] \quad \text{hword}[9] \\
&\quad \text{hword}[10] \quad \text{hword}[11] \\
&\quad \text{hword}[12] \quad \text{hword}[13] \\
&\quad \text{hword}[14] \quad \text{hword}[15] \\
\text{result} &\colon \text{hword}[0] \quad \text{hword}[1] \\
&\quad \text{hword}[2] \quad \text{hword}[3] \\
&\quad \text{hword}[4] \quad \text{hword}[5] \\
&\quad \text{hword}[6] \quad \text{hword}[7] \\
&\quad \text{hword}[8] \quad \text{hword}[9] \\
&\quad \text{hword}[10] \quad \text{hword}[11] \\
&\quad \text{hword}[12] \quad \text{hword}[13] \\
&\quad \text{hword}[14] \quad \text{hword}[15]
\end{align*}
**Vector Add Unsigned Word Modulo VX-form**

\[ \text{vadduwm} \text{ VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3

\[ \text{src1} \leftarrow \text{EXTZ}([\text{VSR}[\text{VRA}+32].\text{word}[i]] \]
\[ \text{src2} \leftarrow \text{EXTZ}([\text{VSR}[\text{VRB}+32].\text{word}[i]]) \]

\[ \text{VSR[VRT+32].word}[i] \leftarrow \text{CHOP32(src1 + src2)} \]
end

For each integer value \( i \) from 0 to 3, do the following.

The integer value in word element \( i \) of \( \text{VSR}[\text{VRA}+32] \) is added to the integer value in word element \( i \) of \( \text{VSR}[\text{VRB}+32] \).

The low-order 32 bits of the result are placed into word element \( i \) of \( \text{VSR}[\text{VRT}+32] \).

**Special Registers Altered:**

None

**Programming Note**

\text{vadduwm} \ can be used for unsigned or signed-integers.

**Register Data Layout for vadduwm**

\begin{align*}
\text{src1} & : \text{VSR}[\text{VRA}+32].\text{word}[0] & \text{VSR}[\text{VRA}+32].\text{word}[1] & \text{VSR}[\text{VRA}+32].\text{word}[2] & \text{VSR}[\text{VRA}+32].\text{word}[3] \\
\text{src2} & : \text{VSR}[\text{VRB}+32].\text{word}[0] & \text{VSR}[\text{VRB}+32].\text{word}[1] & \text{VSR}[\text{VRB}+32].\text{word}[2] & \text{VSR}[\text{VRB}+32].\text{word}[3] \\
\text{result} & : \text{VSR}[\text{VRT}+32].\text{word}[0] & \text{VSR}[\text{VRT}+32].\text{word}[1] & \text{VSR}[\text{VRT}+32].\text{word}[2] & \text{VSR}[\text{VRT}+32].\text{word}[3] \\
\end{align*}

\begin{align*}
0 & \quad 32 & \quad 64 & \quad 48 & \quad 127 \\
\end{align*}

**Vector Add Unsigned Doubleword Modulo VX-form**

\[ \text{vaddudm} \text{ VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>192</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1

\[ \text{src1} \leftarrow \text{EXTZ}([\text{VSR}[\text{VRA}+32].\text{dword}[i]]) \]
\[ \text{src2} \leftarrow \text{EXTZ}([\text{VSR}[\text{VRB}+32].\text{dword}[i]]) \]

\[ \text{VSR[VRT+32].dword}[i] \leftarrow \text{CHOP64(src1 + src2)} \]
end

For each integer value \( i \) from 0 to 1, do the following.

The integer value in doubleword element \( i \) of \( \text{VSR}[\text{VRB}+32] \) is added to the integer value in doubleword element \( i \) of \( \text{VSR}[\text{VRA}+32] \).

The low-order 64 bits of the result are placed into doubleword element \( i \) of \( \text{VSR}[\text{VRT}+32] \).

**Special Registers Altered:**

None

**Programming Note**

\text{vaddudm} \ can be used for signed or unsigned integers.

**Register Data Layout for vaddudm**

\begin{align*}
\text{src1} & : \text{VSR}[\text{VRA}+32].\text{dword}[0] & \text{VSR}[\text{VRA}+32].\text{dword}[1] \\
\text{src2} & : \text{VSR}[\text{VRB}+32].\text{dword}[0] & \text{VSR}[\text{VRB}+32].\text{dword}[1] \\
\text{result} & : \text{VSR}[\text{VRT}+32].\text{dword}[0] & \text{VSR}[\text{VRT}+32].\text{dword}[1] \\
\end{align*}

\begin{align*}
0 & \quad 64 & \quad 127 \\
\end{align*}
Vector Add Unsigned Byte Saturate VX-form

vaddubs VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
src1 ← EXTZ(VSR[VRA+32].byte[i])
src2 ← EXTZ(VSR[VRB+32].byte[i])
VSR[VRT+32].byte[i] ← ui8_CLAMP(src1 + src2)
end

For each integer value i from 0 to 15, do the following.
The unsigned integer value in byte element i of
VSR[VRA+32] is added to the unsigned integer
value in byte element i of VSR[VRB+32].

– If the sum is greater than 2^8-1 the result
saturates to 2^8-1 and SAT is set to 1.

The result is placed into byte element i of
VSR[VRT+32].

Special Registers Altered:
SAT

Register Data Layout for vaddubs

|------------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|

0 8 1 62 43 24 0 4 85 66 47 28 0 8 89 6 1

Vector Add Unsigned Halfword Saturate VX-form

vadduhs VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
src1 ← EXTZ(VSR[VRA+32].hword[i])
src2 ← EXTZ(VSR[VRB+32].hword[i])
VSR[VRT+32].hword[i] ← ui16_CLAMP(src1 + src2)
end

For each integer value i from 0 to 7, do the following.
The unsigned integer value in halfword element i
of VSR[VRA+32] is added to the unsigned integer
value in halfword element i of VSR[VRB+32].

– If the sum is greater than 2^16-1 the result
saturates to 2^16-1 and SAT is set to 1.

The result is placed into halfword element i of
VSR[VRT+32].

Special Registers Altered:
SAT

Register Data Layout for vadduhs

|------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
**Vector Add Unsigned Word Saturate VX-form**

\[
vadduws \quad \text{VRT, VRA, VRB}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>640</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()|

\[
\text{do } i = 0 \text{ to } 3 \\
\quad \text{src1 } \leftarrow \text{EXTZ(VSR[VRT+32].word[i])} \\
\quad \text{src2 } \leftarrow \text{EXTZ(VSR[VRT+32].word[i])} \\
\quad \text{VSR[VRT+32].word[i] } \leftarrow \text{ui32_CLAMP(src1 + src2)}
\]

For each integer value \( i \) from 0 to 3, do the following.

The unsigned integer value in word element \( i \) of \( \text{VSR[VRT+32]} \) is added to the unsigned integer value in word element \( i \) of \( \text{VSR[VRT+32]} \).

- If the sum is greater than \( 2^{32}-1 \) the result saturates to \( 2^{32}-1 \) and SAT is set to 1.

The result is placed into word element \( i \) of \( \text{VSR[VRT+32]} \).

**Special Registers Altered:**

SAT

**Register Data Layout for vadduws**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>48</td>
<td>127</td>
</tr>
</tbody>
</table>
Vector Add Unsigned Quadword Modulo VX-form

\[ \text{vadduqm} \ VRT, VRA, VRB \]

\[
\begin{array}{cccccc}
0 & 6 & 11 & 16 & 21 & 256 \\
\end{array}
\]

\[
\begin{array}{cccccc}
4 & VRT & VRA & VRB & 256 \\
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(VSR[VRB+32])

VSR[VRT+32] ← CHOP128(src1 + src2)

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].

src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1 and src2 are placed into VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vadduqm

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
</table>

Vector Add Extended Unsigned Quadword Modulo VA-form

\[ \text{vaddeuqm} \ VRT, VRA, VRB, VRC \]

\[
\begin{array}{cccccc}
0 & 6 & 11 & 16 & 21 & 256 \\
\end{array}
\]

\[
\begin{array}{cccccc}
4 & VRT & VRA & VRB & VRC & 256 \\
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(VSR[VRB+32])
cin ← EXTZ(VSR[VRC+32].bit[127])

VSR[VRT+32] ← CHOP128(src1 + src2 + cin)

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].
Let cin be the integer value in bit 127 of VSR[VRC+32].

src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1, src2, and cin are placed into VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vaddeuqm

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>result</th>
</tr>
</thead>
</table>
Vector Add & write Carry-out Unsigned Quadword VX-form

```
vaddcuq VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>320</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>
```

if MSR.VEC=0 then Vector_Unavailable();

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(VSR[VRB+32])
sum ← EXTZ(src1) + EXTZ(src2)
VSR[VRT+32] ← EXTZ128((src1 + src2) >> 128)

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].

src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1 and src2 is placed into VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vaddcuq

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>result</th>
</tr>
</thead>
</table>

Vector Add Extended & write Carry-out Unsigned Quadword VA-form

```
vaddecuq VRT,VRA,VRB,VRC

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>
```

if MSR.VEC=0 then Vector_Unavailable();

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(VSR[VRB+32])
cin ← EXTZ(VSR[VRC+32].bit[127])
VSR[VRT+32] ← EXTZ128((src1 + src2 + cin) >> 128)

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].
Let cin be the integer value in bit 127 of VSR[VRC+32].

src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1, src2, and cin are placed into VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vaddecuq

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>result</th>
</tr>
</thead>
</table>

Programming Note

The Vector Add Unsigned Quadword instructions support efficient wide-integer addition. The following code sequence can be used to implement a 512-bit signed or unsigned add operation.

```
vadduqm vS3,vA3,vB3 # bits 384:511 of sum
vaddcuq vC3,vA3,vB3 # carry out of bit 384 of sum
vaddeuqm vS2,vA2,vB2,vC3 # bits 256:383 of sum
vaddecuq vC2,vA2,vB2,vC3 # carry out of bit 256 of sum
vaddeuqm vS1,vA1,vB1,vC2 # bits 128:255 of sum
vaddecuq vC1,vA1,vB1,vC2 # carry out of bit 128 of sum
vaddeuqm vS0,vA0,vB0,vC1 # bits 0:127 of sum
```
6.9.1.2 Vector Integer Subtract Instructions

**Vector Subtract & Write Carry-Out Unsigned Word VX-form**

Vector Subtract & Write Carry-Out Unsigned Word VX-form

```plaintext
vsubcuw VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
src1 ← EXTZ(VSR[VRA+32].word[i])
src2 ← EXTZ(~VSR[VRB+32].word[i])

VSR[VRT+32].word[i] ← EXTZ32((src1+src2+1) >> 32)
end

For each integer value i from 0 to 3, do the following.
The unsigned integer value in word element i of VSR[VRB+32] is subtracted from the unsigned integer value in word element i in VSR[VRA+32].
The complement of the borrow out of bit 0 of the 32-bit difference is zero-extended to 32 bits and placed into word element i of VSR[VRT+32].

**Special Registers Altered:**
None

**Vector Subtract Signed Byte Saturate VX-form**

Vector Subtract Signed Byte Saturate VX-form

```plaintext
vsubsbs VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
src1 ← EXTS(VSR[VRA+32].byte[i])
src2 ← EXTS(VSR[VRB+32].byte[i])

VSR[VRT+32].byte[i] ← si8_CLAMP(src1 + ¬src2 + 1)
end

For each integer value i from 0 to 15, do the following.
The signed integer value in byte element i in VSR[VRB+32] is subtracted from the signed integer value in byte element i in VSR[VRA+32].

- If the intermediate result is greater than 127 the result saturates to 127 and SAT is set to 1.
- If the intermediate result is less than -128 the result saturates to -128 and SAT is set to 1.

The result is placed into byte element i of VSR[VRT+32].

**Special Registers Altered:**
SAT

**Register Data Layout for vsubcuw**

|------|---------------------|---------------------|---------------------|---------------------|

**Register Data Layout for vsubsbs**

|------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
Vector Subtract Signed Halfword Saturate
VX-form

\[ \text{vsubshs} \text{ VRT, VRA, VRB} \]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 1856
\end{array}
\]

\begin{align*}
\text{if } & \text{ MSR.VEC=0 then Vector Unavailable()}
\end{align*}

\begin{align*}
\text{do } i &= 0 \text{ to } 7 \\
\text{src1} &\leftarrow \text{EXTS(VSR}[\text{VRA}+32].\text{hword}[i]) \\
\text{src2} &\leftarrow \text{EXTS(VSR}[\text{VRB}+32].\text{hword}[i]) \\
\text{VSR}[\text{VRT}+32].\text{hword}[i] &\leftarrow \text{si16_CLAMP(src1 + ¬src2 + 1)}
\end{align*}

\begin{align*}
\text{end}
\end{align*}

For each integer value \( i \) from 0 to 7, do the following.

The signed integer value in halfword element \( i \) in VSR[VRB+32] is subtracted from the signed integer value in halfword element \( i \) in VSR[VRA+32].

\begin{itemize}
\item If the intermediate result is greater than \( 2^{15}-1 \), the result saturates to \( 2^{15}-1 \) and SAT is set to 1.
\item If the intermediate result is less than \( -2^{15} \) the result saturates to \( -2^{15} \) and SAT is set to 1
\end{itemize}

The result is placed into halfword element \( i \) of VSR[VRT+32].

Special Registers Altered:

SAT

Register Data Layout for vsubshs

\[
\begin{array}{cccccccc}
src1 & VSR[\text{VRA}+32].\text{hword}[0] & VSR[\text{VRA}+32].\text{hword}[1] & VSR[\text{VRA}+32].\text{hword}[2] & VSR[\text{VRA}+32].\text{hword}[3] & VSR[\text{VRA}+32].\text{hword}[4] & VSR[\text{VRA}+32].\text{hword}[5] & VSR[\text{VRA}+32].\text{hword}[6] & VSR[\text{VRA}+32].\text{hword}[7] \\
result & VSR[\text{VRT}+32].\text{hword}[0] & VSR[\text{VRT}+32].\text{hword}[1] & VSR[\text{VRT}+32].\text{hword}[2] & VSR[\text{VRT}+32].\text{hword}[3] & VSR[\text{VRT}+32].\text{hword}[4] & VSR[\text{VRT}+32].\text{hword}[5] & VSR[\text{VRT}+32].\text{hword}[6] & VSR[\text{VRT}+32].\text{hword}[7]
\end{array}
\]

Vector Subtract Signed Word Saturate
VX-form

\[ \text{vsubsws} \text{ VRT, VRA, VRB} \]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 1920
\end{array}
\]

\begin{align*}
\text{if } & \text{ MSR.VEC=0 then Vector Unavailable()}
\end{align*}

\begin{align*}
\text{do } i &= 0 \text{ to } 3 \\
\text{src1} &\leftarrow \text{EXTS(VSR}[\text{VRA}+32].\text{word}[i]) \\
\text{src2} &\leftarrow \text{EXTS(VSR}[\text{VRB}+32].\text{word}[i]) \\
\text{VSR}[\text{VRT}+32].\text{word}[i] &\leftarrow \text{si32_CLAMP(src1 + ¬src2 + 1)}
\end{align*}

\begin{align*}
\text{end}
\end{align*}

For each integer value \( i \) from 0 to 3, do the following.

The signed integer value in word element \( i \) in VSR[VRB+32] is subtracted from the signed integer value in word element \( i \) in VSR[VRA+32].

\begin{itemize}
\item If the intermediate result is greater than \( 2^{31}-1 \), the result saturates to \( 2^{31}-1 \) and SAT is set to 1.
\item If the intermediate result is less than \( -2^{31} \) the result saturates to \( -2^{31} \) and SAT is set to 1
\end{itemize}

The result is placed into word element \( i \) of VSR[VRT+32].

Special Registers Altered:

SAT

Register Data Layout for vsubsws

\[
\begin{array}{cccccccc}
src1 & VSR[\text{VRA}+32].\text{word}[0] & VSR[\text{VRA}+32].\text{word}[1] & VSR[\text{VRA}+32].\text{word}[2] & VSR[\text{VRA}+32].\text{word}[3] \\
src2 & VSR[\text{VRB}+32].\text{word}[0] & VSR[\text{VRB}+32].\text{word}[1] & VSR[\text{VRB}+32].\text{word}[2] & VSR[\text{VRB}+32].\text{word}[3] \\
result & VSR[\text{VRT}+32].\text{word}[0] & VSR[\text{VRT}+32].\text{word}[1] & VSR[\text{VRT}+32].\text{word}[2] & VSR[\text{VRT}+32].\text{word}[3]
\end{array}
\]
Vector Subtract Unsigned Byte Modulo

**VX-form**

\[ \text{vsububm} \ VRT,VRA,VRB \]

1. If MSR.VEC = 0 then Vector_Unavailable()
2. For each integer value \( i \) from 0 to 15, do the following.
   - The unsigned integer value in byte element \( i \) in VSR[VRB+32] is subtracted from the unsigned integer value in byte element \( i \) in VSR[VRA+32].
   - The low-order 8 bits of the result are placed into byte element \( i \) of VSR[VRT+32].

**Special Registers Altered:** None

**Register Data Layout for vsububm**

|------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|

Vector Subtract Unsigned Halfword Modulo

**VX-form**

\[ \text{vsubuhm} \ VRT,VRA,VRB \]

1. If MSR.VEC = 0 then Vector_Unavailable()
2. For each integer value \( i \) from 0 to 7, do the following.
   - The unsigned integer value in halfword element \( i \) in VSR[VRB+32] is subtracted from the unsigned integer value in halfword element \( i \) in VSR[VRA+32].
   - The low-order 16 bits of the result are placed into halfword element \( i \) of VSR[VRT+32].

**Special Registers Altered:** None

**Register Data Layout for vsubuhm**

|------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|
**Vector Subtract Unsigned Word Modulo VX-form**

### vsubuwm

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1152</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

`vsubuwm VRT, VRA, VRB`

```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 3
    src1 ← EXTZ(VSR[VRB+32].word[i])
    src2 ← EXTZ(VSR[VRA+32].word[i])
    VSR[VRT+32].word[i] ← CHOP32(src1 + ¬src2 + 1)
end
```

For each integer value `i` from 0 to 3, do the following.

- The unsigned integer value in word element `i` in `VSR[VRB+32]` is subtracted from the unsigned integer value in word element `i` in `VSR[VRA+32]`.
- The low-order 16 bits of the result are placed into word element `i` of `VSR[VRT+32]`.

### Special Registers Altered:
- None

**Register Data Layout for vsubuwm**

|------|---------------------|---------------------|---------------------|---------------------|

**Vector Subtract Unsigned Doubleword Modulo VX-form**

### vsubudm

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1216</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>21</td>
</tr>
</tbody>
</table>

`vsubudm VRT, VRA, VRB`

```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 1
    src1 ← EXTZ(VSR[VRA+32].dword[i])
    src2 ← EXTZ(VSR[VRB+32].dword[i])
    VSR[VRT+32].dword[i] ← CHOP64(src1 + ¬src2 + 1)
end
```

For each integer value `i` from 0 to 1, do the following.

- The integer value in doubleword element `i` in `VSR[VRB+32]` is subtracted from the integer value in doubleword element `i` in `VSR[VRA+32]`.
- The low-order 64 bits of the result are placed into doubleword element `i` of `VSR[VRT+32]`.

### Special Registers Altered:
- None

**Programming Note**

`vsubudm` can be used for signed or unsigned integers.

**Register Data Layout for vsubudm**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
</tbody>
</table>
### Vector Subtract Unsigned Byte Saturate

**VX-form**

VX-form

```plaintext
vsububs VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
    src1 ← EXTZ(VSR[VRA+32].byte[i])
    src2 ← EXTZ(VSR[VRB+32].byte[i])
    VSR[VRT+32].byte[i] ← ui8_CLAMP(src1 + ¬src2 + 1)
end
```

For each integer value \(i\) from 0 to 15, do the following.

- The unsigned integer value in byte element \(i\) of \(VSR[VRB+32]\) is subtracted from the unsigned integer value in byte element \(i\) of \(VSR[VRA+32]\).
- If the intermediate result is less than 0 the result saturates to 0 and \(SAT\) is set to 1.

The result is placed into byte element \(i\) of \(VSR[VRT+32]\).

**Special Registers Altered:**

\(SAT\)

**Register Data Layout for vsububs**

|-------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|

**Register Data Layout for vsubuh**

|-------|----------|----------|----------|----------|----------|----------|----------|----------|
Vector Subtract Unsigned Word Saturate

**VX-form**

\[ vsubuws \ VRT, VRA, VRB \]

- If MSR.VEC=0 then Vector_Unavailable()
- \[ \text{do } i = 0 \text{ to } 3 \]
  - \[ \text{src1} \leftarrow \text{EXTZ(VSR}[VRA+32].\text{word}[i]) \]
  - \[ \text{src2} \leftarrow \text{EXTZ(VSR}[VRB+32].\text{word}[i]) \]
  - \[ \text{VSR}[VRT+32].\text{word}[i] \leftarrow \text{ui32_CLAMP(src1 + ¬src2 + 1)} \]
- \[ \text{end} \]

For each integer value \( i \) from 0 to 7, do the following.
- The unsigned integer value in word element \( i \) of \( \text{VSR}[VRB+32] \) is subtracted from the unsigned integer value in word element \( i \) of \( \text{VSR}[VRA+32] \).
  - If the intermediate result is less than 0 the result saturates to 0 and \( \text{SAT} \) is set to 1.

The result is placed into word element \( i \) of \( \text{VSR}[VRT+32] \).

**Special Registers Altered:**
- \( \text{SAT} \)

**Register Data Layout for vsubuws**

|------|---------------------|---------------------|---------------------|---------------------|
**Vector Subtract Unsigned Quadword Modulo VX-form**

\[ vsubuqm \quad VRT,VRA,VRB \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>1280</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(VSR[VRB+32])

VSR[VRT+32] ← CHOP128(src1 + src2 + 1)

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].

src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1, the one's complement of src2, and the value 1 are placed into VSR[VRT+32].

**Special Registers Altered:**
None

---

**Register Data Layout for vsubuq**

- src1 \( \rightarrow \) VSR[VRA+32]
- src2 \( \rightarrow \) VSR[VRB+32]
- result \( \rightarrow \) VSR[VRT+32]

**Register Data Layout for vsubeuq**

- src1 \( \rightarrow \) VSR[VRA+32]
- src2 \( \rightarrow \) VSR[VRB+32]
- src3 \( \rightarrow \) VSR[VRC+32]
- result \( \rightarrow \) VSR[VRT+32]
Vector Subtract & write Carry-out Unsigned Quadword VX-form

\[ \text{vsubcuq} \quad \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1344</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[ \text{src1} \leftarrow \text{EXTZ}(\text{VSR}[\text{VRA}+32]) \]
\[ \text{src2} \leftarrow \text{EXTZ}(\text{VSR}[\text{VRB}+32]) \]
\[ \text{VSR}[\text{VRT}+32] \leftarrow \text{CHOP128}(\text{src1} + \text{src2} + 1) \gg 128 \]

Let \( \text{src1} \) be the integer value in \( \text{VSR}[\text{VRA}+32] \).
Let \( \text{src2} \) be the integer value in \( \text{VSR}[\text{VRB}+32] \).

\( \text{src1} \) and \( \text{src2} \) can be signed or unsigned integers.

The carry out of the sum of \( \text{src1} \), the one’s complement of \( \text{src2} \), and the value 1 is placed into \( \text{VSR}[\text{VRT}+32] \).

Special Registers Altered:
None

Register Data Layout for vsubcuq

- **src1**: \( \text{VSR}[\text{VRA}+32] \)
- **src2**: \( \text{VSR}[\text{VRB}+32] \)
- **result**: \( \text{VSR}[\text{VRT}+32] \)

Vector Subtract Extended & write Carry-out Unsigned Quadword VA-form

\[ \text{vsubecuq} \quad \text{VRT, VRA, VRB, VRC} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[ \text{src1} \leftarrow \text{EXTZ}(\text{VSR}[\text{VRA}+32]) \]
\[ \text{src2} \leftarrow \text{EXTZ}(\text{VSR}[\text{VRB}+32]) \]
\[ \text{cin} \leftarrow \text{EXTZ}(\text{VSR}[\text{VRC}+32].\text{bit}[127]) \]
\[ \text{VSR}[\text{VRT}+32] \leftarrow \text{CHOP128}((\text{src1} + \text{src2} + \text{cin}) \gg 128) \]

Let \( \text{src1} \) be the integer value in \( \text{VSR}[\text{VRA}+32] \).
Let \( \text{src2} \) be the integer value in \( \text{VSR}[\text{VRB}+32] \).
Let \( \text{cin} \) be the integer value in bit 127 of \( \text{VSR}[\text{VRC}+32] \).

\( \text{src1} \) and \( \text{src2} \) can be signed or unsigned integers.

The carry out of the sum of \( \text{src1} \), the one’s complement of \( \text{src2} \), and \( \text{cin} \) are placed into \( \text{VSR}[\text{VRT}+32] \).

Special Registers Altered:
None

Register Data Layout for vsubecuq

- **src1**: \( \text{VSR}[\text{VRA}+32] \)
- **src2**: \( \text{VSR}[\text{VRB}+32] \)
- **src3**: \( \text{VSR}[\text{VRC}+32] \)
- **result**: \( \text{VSR}[\text{VRT}+32] \)

Programming Note

The **Vector Subtract Unsigned Quadword** instructions support efficient wide-integer subtraction. The following code sequence can be used to implement a 512-bit signed or unsigned subtract operation.

\[
\begin{align*}
\text{vsubuqm} & \quad \text{vS3, vA3, vB3} \quad \# \text{bits 384:511 of difference} \\
\text{vsubcuq} & \quad \text{vC3, vA3, vB3} \quad \# \text{carry out of bit 384 of difference} \\
\text{vsubuqm} & \quad \text{vS2, vA2, vB2, vC3} \quad \# \text{bits 256:383 of difference} \\
\text{vsubcuq} & \quad \text{vC2, vA2, vB2, vC3} \quad \# \text{carry out of bit 256 of difference} \\
\text{vsubuqm} & \quad \text{vS1, vA1, vB1, vC2} \quad \# \text{bits 128:255 of difference} \\
\text{vsubcuq} & \quad \text{vC1, vA1, vB1, vC2} \quad \# \text{carry out of bit 128 of difference} \\
\text{vsubuqm} & \quad \text{vS0, vA0, vB0, vC1} \quad \# \text{bits 0:127 of difference}
\end{align*}
\]
6.9.1.3 Vector Integer Multiply Instructions

Vector Multiply Even Signed Byte VX-form

vmulesb VRT, VRA, VRB

| 4 | 8 | 11 | 16 | 21 | 776 |

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 7
    src1 ← EXTS(VSR[VRA+32].byte[2×i])
    src2 ← EXTS(VSR[VRB+32].byte[2×i])
    VSR[VRT+32].hword[i] ← CHOP16(src1 × src2)
end

For each integer value i from 0 to 7, do the following. The signed integer value in byte element i×2 of VSR[VRA+32] is multiplied by the signed integer value in byte element i×2 of VSR[VRB+32].

The 16-bit product is placed into halfword element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vmulesb

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
</tr>
</tbody>
</table>

Vector Multiply Odd Signed Byte VX-form

vmulosb VRT, VRA, VRB

| 4 | 8 | 11 | 16 | 21 | 264 |

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 7
    src1 ← EXTS(VSR[VRA+32].byte[2×i+1])
    src2 ← EXTS(VSR[VRB+32].byte[2×i+1])
    VSR[VRT+32].hword[i] ← CHOP16(src1 × src2)
end

For each integer value i from 0 to 7, do the following. The signed integer value in byte element i×2+1 of VSR[VRA+32] is multiplied by the signed integer value in byte element i×2+1 of VSR[VRB+32].

The 16-bit product is placed into halfword element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vmulosb

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
</tr>
</tbody>
</table>
Vector Multiply Even Unsigned Byte VX-form

`vmuleub VRT, VRA, VRB`

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>520</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC = 0 then Vector_Unavailable()

Do i = 0 to 7
src1 ← EXTZ(VSR[VRA+32].byte[2×i])
src2 ← EXTZ(VSR[VRB+32].byte[2×i])
VSR[VRT+32].hword[i] ← CHOP16(src1 × src2)

End

For each integer value i from 0 to 7, do the following.
The unsigned integer value in byte element 2×i of VSR[VRA+32] is multiplied by the unsigned integer value in byte element 2×i of VSR[VRB+32].

The 16-bit product is placed into halfword element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for `vmuleub`

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[0]</td>
<td>unused</td>
<td>VSR[VRT+32].hword[0]</td>
</tr>
</tbody>
</table>

Vector Multiply Odd Unsigned Byte VX-form

`vmuloub VRT, VRA, VRB`

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC = 0 then Vector_Unavailable()

Do i = 0 to 7
src1 ← EXTZ(VSR[VRA+32].byte[2×i+1])
src2 ← EXTZ(VSR[VRB+32].byte[2×i+1])
VSR[VRT+32].hword[i] ← CHOP16(src1 × src2)

End

For each integer value i from 0 to 7, do the following.
The unsigned integer value in byte element 2×i+1 of VSR[VRA+32] is multiplied by the unsigned integer value in byte element 2×i+1 of VSR[VRB+32].

The 16-bit product is placed into halfword element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for `vmuloub`

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>byte[1]</td>
<td>VSR[VRT+32].hword[0]</td>
</tr>
</tbody>
</table>
Vector Multiply Even Signed Halfword VX-form

**vmulesh**

\[
\begin{align*}
&\text{VX-form} \\
&\text{vmulesh} \quad \text{VRT, VRA, VRB} \\
&\begin{array}{cccc}
4 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\end{align*}
\]

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3

\[
\begin{align*}
&\text{src1} \leftarrow \text{EXTS(VSR[VRA+32].hword}[2\times i]) \\
&\text{src2} \leftarrow \text{EXTS(VSR[VRB+32].hword}[2\times i]) \\
&\text{VSR}[VRT+32].\text{word}[i] \leftarrow \text{CHOP32(src1 \times src2)}
\end{align*}
\]

done

For each integer value \( i \) from 0 to 3, do the following.
The signed integer value in halfword element \( i \times 2 \) of \( \text{VSR}[\text{VRA+32}] \) is multiplied by the signed integer value in halfword element \( i \times 2 \) of \( \text{VSR}[\text{VRB+32}] \).
The 32-bit product is placed into word element \( i \) of \( \text{VSR}[\text{VRT+32}] \).

Special Registers Altered:
None

**Register Data Layout for vmulesh**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VR+32].hword[0]</td>
<td>unused</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
</tbody>
</table>

Vector Multiply Odd Signed Halfword VX-form

**vmulosh**

\[
\begin{align*}
&\text{VX-form} \\
&\text{vmulosh} \quad \text{VRT, VRA, VRB} \\
&\begin{array}{cccc}
4 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\end{align*}
\]

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3

\[
\begin{align*}
&\text{src1} \leftarrow \text{EXTS(VSR[VRA+32].hword}[2\times i+1]) \\
&\text{src2} \leftarrow \text{EXTS(VSR[VRB+32].hword}[2\times i+1]) \\
&\text{VSR}[VRT+32].\text{word}[i] \leftarrow \text{CHOP32(src1 \times src2)}
\end{align*}
\]

done

For each integer value \( i \) from 0 to 3, do the following.
The signed integer value in halfword element \( i \times 2+1 \) of \( \text{VSR}[\text{VRA+32}] \) is multiplied by the signed integer value in halfword element \( i \times 2+1 \) of \( \text{VSR}[\text{VRB+32}] \).
The 32-bit product is placed into word element \( i \) of \( \text{VSR}[\text{VRT+32}] \).

Special Registers Altered:
None

**Register Data Layout for vmulosh**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>VSR[VR+32].hword[1]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
</tbody>
</table>
Vector Multiply Even Unsigned Halfword VX-form

**vmuleuh**  

**VRT, VRA, VRB**

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src1 ← EXTZ(VSR[VRA+32].hword[2×i])
    src2 ← EXTZ(VSR[VRB+32].hword[2×i])
    VSR[VRT+32].word[i] ← CHOP32(src1 × src2)
end
```

For each integer value \( i \) from 0 to 3, do the following.

The unsigned integer value in halfword element \( i \times 2 \) of VSR[VRA+32] is multiplied by the unsigned integer value in halfword element \( i \times 2 \) of VSR[VRB+32].

The 32-bit product is placed into word element \( i \) of VSR[VRT+32].

Special Registers Altered:

None

---

Register Data Layout for vmuleuh

|------------|--------|----------------------|--------|----------------------|--------|----------------------|--------|----------------------|--------|
```

---

Register Data Layout for vmulouh

|------------|--------|----------------------|--------|----------------------|--------|----------------------|--------|----------------------|--------|

---

Vector Multiply Odd Unsigned Halfword VX-form

**vmulouh**  

**VRT, VRA, VRB**

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src1 ← EXTZ(VSR[VRA+32].hword[2×i+1])
    src2 ← EXTZ(VSR[VRB+32].hword[2×i+1])
    VSR[VRT+32].word[i] ← CHOP32(src1 × src2)
end
```

For each integer value \( i \) from 0 to 3, do the following.

The unsigned integer value in halfword element \( i \times 2+1 \) of VSR[VRA+32] is multiplied by the unsigned integer value in halfword element \( i \times 2+1 \) of VSR[VRB+32].

The 32-bit product is placed into word element \( i \) of VSR[VRT+32].

Special Registers Altered:

None
Vector Multiply Even Signed Word VX-form

\[ \text{vmulesw} \ VRT,VRA,VRB \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>VRT</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>904</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();
do \( i = 0 \) to 1
\[
\begin{align*}
src1 & \leftarrow \text{EXTS}(\text{VSR}[VRA+32].\text{word}[2i]) \\
src2 & \leftarrow \text{EXTS}(\text{VSR}[VRB+32].\text{word}[2i]) \\
\end{align*}
\]
VSR[VRT+32].dword[i] \leftarrow \text{CHOP64}(src1 \times src2)
end

For each integer value \( i \) from 0 to 1, do the following.
The signed integer in word element \( 2i \) of VSR[VRA+32] is multiplied by the signed integer in word element \( 2i \) of VSR[VRB+32].
The 64-bit product is placed into doubleword element \( i \) of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vmulesw

\[
\begin{array}{cccc}
\text{src1} & \text{VSR}[VRA+32].\text{word}[0] & \text{unused} & \text{VSR}[VRA+32].\text{word}[2] & \text{unused} \\
\text{src2} & \text{VSR}[VRB+32].\text{word}[0] & \text{unused} & \text{VSR}[VRB+32].\text{word}[2] & \text{unused} \\
\text{result} & \text{VSR}[VRT+32].\text{dword}[0] & \text{VSR}[VRT+32].\text{dword}[1] & & \\
\end{array}
\]

Vector Multiply Odd Signed Word VX-form

\[ \text{vmulosw} \ VRT,VRA,VRB \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>VRT</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>392</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();
do \( i = 0 \) to 1
\[
\begin{align*}
src1 & \leftarrow \text{EXTS}(\text{VSR}[VRA+32].\text{word}[2i+1]) \\
src2 & \leftarrow \text{EXTS}(\text{VSR}[VRB+32].\text{word}[2i+1]) \\
\end{align*}
\]
VSR[VRT+32].dword[i] \leftarrow \text{CHOP64}(src1 \times src2)
end

For each integer value \( i \) from 0 to 1, do the following.
The signed integer in word element \( 2i + 1 \) of VSR[VRA+32] is multiplied by the signed integer in word element \( 2i + 1 \) of VSR[VRB+32].
The 64-bit product is placed into doubleword element \( i \) of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vmulosw

\[
\begin{array}{cccc}
\text{src1} & \text{unused} & \text{VSR}[VRA+32].\text{word}[1] & \text{unused} & \text{VSR}[VRA+32].\text{word}[3] \\
\text{src2} & \text{unused} & \text{VSR}[VRB+32].\text{word}[1] & \text{unused} & \text{VSR}[VRB+32].\text{word}[3] \\
\text{result} & \text{VSR}[VRT+32].\text{dword}[0] & \text{VSR}[VRT+32].\text{dword}[1] & & \\
\end{array}
\]
Vector Multiply Even Unsigned Word VX-form

vmuleuw VRT, VRA, VRB

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 1
  src1 ← EXTZ(VSR[VRA+32].word[2×i])
  src2 ← EXTZ(VSR[VRB+32].word[2×i])
  VSR[VRT+32].dword[i] ← CHOP64(src1 × src2)
end

For each integer value \( i \) from 0 to 1, do the following.
The unsigned integer in word element \( 2 \times i \) of VSR[VRA+32] is multiplied by the unsigned integer in word element \( 2 \times i \) of VSR[VRB+32].

The 64-bit product is placed into doubleword element \( i \) of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vmuleuw

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>unused</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
<tr>
<td>unused</td>
<td>unused</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
</tbody>
</table>

Register Data Layout for vmulouw

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>unused</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
<tr>
<td>unused</td>
<td>unused</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
</tbody>
</table>

Vector Multiply Odd Unsigned Word VX-form

vmulouw VRT, VRA, VRB

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 1
  src1 ← EXTZ(VSR[VRA+32].word[2×i+1])
  src2 ← EXTZ(VSR[VRB+32].word[2×i+1])
  VSR[VRT+32].dword[i] ← CHOP64(src1 × src2)
end

For each integer value \( i \) from 0 to 1, do the following.
The unsigned integer in word element \( 2 \times i + 1 \) of VSR[VRA+32] is multiplied by the unsigned integer in word element \( 2 \times i + 1 \) of VSR[VRB+32].

The 64-bit product is placed into doubleword element \( i \) of VSR[VRT+32].

Special Registers Altered:
None
Vector Multiply Even Unsigned Doubleword

**VX-form**

```
vmuleud VRT,VRA,VRB
```

If MSR.VEC=0 then Vector_Unavailable().

```
src1 ← EXTZ(VSR[VRA+32].dword[0])
src2 ← EXTZ(VSR[VRB+32].dword[0])
VSR[VRT+32] ← CHOP128(src1 × src2)
```

Let src1 be the unsigned integer value in doubleword element 0 of VSR[VRA+32].

Let src2 be the unsigned integer value in doubleword element 0 of VSR[VRB+32].

The 128-bit product of src1 multiplied by src2 is placed into VSR[VRT+32].

**Special Registers Altered:**

None

---

Vector Multiply Odd Unsigned Doubleword

**VX-form**

```
vmuloud VRT,VRA,VRB
```

If MSR.VEC=0 then Vector_Unavailable().

```
src1 ← EXTZ(VSR[VRA+32].dword[1])
src2 ← EXTZ(VSR[VRB+32].dword[1])
VSR[VRT+32] ← CHOP128(src1 × src2)
```

Let src1 be the unsigned integer value in doubleword element 1 of VSR[VRA+32].

Let src2 be the unsigned integer value in doubleword element 1 of VSR[VRB+32].

The 128-bit product of src1 multiplied by src2 is placed into VSR[VRT+32].

**Special Registers Altered:**

None

---

**Register Data Layout for vmuleud**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRA+32].dword[0]</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

**Register Data Layout for vmuloud**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>VSR[VRA+32].dword[1]</td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>unused</td>
<td>VSR[VRB+32].dword[1]</td>
<td></td>
</tr>
</tbody>
</table>
Vector Multiply Even Signed Doubleword VX-form

vmulesd  VRT,VRA,VRB

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>968</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then VectorUnavailable()

src1 ← ExTS(VSR[VRA+32].dword[0])

src2 ← ExTS(VSR[VRB+32].dword[0])

VSR[VRT+32] ← CHOP128(src1 × src2)

Let src1 be the signed integer value in doubleword element 0 of VSR[VRA+32].

Let src2 be the signed integer value in doubleword element 0 of VSR[VRB+32].

The 128-bit product of src1 multiplied by src2 is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmulesd

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>

Vector Multiply Odd Signed Doubleword VX-form

vmulosd  VRT,VRA,VRB

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>456</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then VectorUnavailable()

src1 ← ExTS(VSR[VRA+32].dword[1])

src2 ← ExTS(VSR[VRB+32].dword[1])

VSR[VRT+32] ← CHOP128(src1 × src2)

Let src1 be the signed integer value in doubleword element 1 of VSR[VRA+32].

Let src2 be the signed integer value in doubleword element 1 of VSR[VRB+32].

The 128-bit product of src1 multiplied by src2 is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmulosd

<table>
<thead>
<tr>
<th>src1</th>
<th>unused</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>unused</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>
Vector Multiply Unsigned Word Modulo VX-form

vmuluwm  VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src1 ← EXTZ(VSR[VRA+32].word[i])
    src2 ← EXTZ(VSR[VRB+32].word[i])
    VSR[VRT+32].word[i] ← CHOP32(src1 × src2)
end

For each integer value \(i\) from 0 to 3, do the following.

The integer in word element \(i\) of VSR[VRA+32] is multiplied by the integer in word element \(i\) of VSR[VRB+32].

The low-order 32 bits of the product are placed into word element \(i\) of VSR[VRT+32].

Special Registers Altered:
None

---

Programming Note

vmuluwm can be used for unsigned or signed integers.

---

Register Data Layout for vmuluwm

|------|---------------------|---------------------|---------------------|---------------------|

Vector Multiply High Signed Word VX-form

vmulhsw  VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src1 ← EXTS(VSR[VRA+32].word[i])
    src2 ← EXTS(VSR[VRB+32].word[i])
    VSR[VRT+32].word[i] ← CHOP32((src1 × src2) >> 32)
end

For each integer value \(i\) from 0 to 3, do the following.

The signed integer value in word element \(i\) of VSR[VRA+32] is multiplied by the signed integer value in word element \(i\) of VSR[VRB+32].

The high-order 32 bits of the 64-bit product are placed into word element \(i\) of VSR[VRT+32].

Special Registers Altered:
None

---

Register Data Layout for vmulhsw

|------|---------------------|---------------------|---------------------|---------------------|
Vector Multiply High Unsigned Word VX-form

vmulhuw VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable||

\begin{align*}
\text{do } & i = 0 \text{ to } 3 \\
& \text{src1 } \leftarrow \text{EXTZ(VSR[VRA+32].word[i])} \\
& \text{src2 } \leftarrow \text{EXTZ(VSR[VRB+32].word[i])} \\
& \text{VSR[VRT+32].word[i]} \leftarrow \text{CHOP32((src1 \times src2) >> 32)} \\
\end{align*}

end

For each integer value \( i \) from 0 to 3, do the following.

The unsigned integer value in word element \( i \) of VSR[VRA+32] is multiplied by the unsigned integer value in word element \( i \) of VSR[VRB+32].

The high-order 32 bits of the 64-bit product are placed into word element \( i \) of VSR[VRT+32].

Special Registers Altered:
None

<table>
<thead>
<tr>
<th>Register Data Layout for vmulhuw</th>
</tr>
</thead>
</table>
Vector Multiply High Signed Doubleword

**VX-form**

\[ \text{vmulhsd \ VRT,VRA,VRB} \]

If MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 1
  src1 ← EXTS(VSR[VRA+32].dword[i])
  src2 ← EXTS(VSR[VRB+32].dword[i])

  VSR[VRT+32].dword[i] ← CHOP64((src1 × src2) >> 64)
end

For each integer value \( i \) from 0 to 1, do the following.

The signed integer value in doubleword element \( i \) of VSR[VRA+32] is multiplied by the signed integer value in doubleword element \( i \) of VSR[VRB+32].

The high-order 64 bits of the 128-bit product are placed into doubleword element \( i \) of VSR[VRT+32].

**Special Registers Altered:** None

**Register Data Layout for vmulhsd**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
</tbody>
</table>

Vector Multiply High Unsigned Doubleword

**VX-form**

\[ \text{vmulhud \ VRT,VRA,VRB} \]

If MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 1
  src1 ← EXTZ(VSR[VRA+32].dword[i])
  src2 ← EXTZ(VSR[VRB+32].dword[i])

  VSR[VRT+32].dword[i] ← CHOP64((src1 × src2) >> 64)
end

For each integer value \( i \) from 0 to 1, do the following.

The unsigned integer value in doubleword element \( i \) of VSR[VRA+32] is multiplied by the unsigned integer value in doubleword element \( i \) of VSR[VRB+32].

The high-order 64 bits of the 128-bit product are placed into doubleword element \( i \) of VSR[VRT+32].

**Special Registers Altered:** None

**Register Data Layout for vmulhud**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
</tbody>
</table>
Vector Multiply Low Doubleword VX-form

vmulld VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
    src1 ← EKTS(VSR[VRA+32].dword[i])
    src2 ← EKTS(VSR[VRB+32].dword[i])
    VSR[VRT+32].dword[i] ← CHOP64(src1 × src2)
end

For each integer value i from 0 to 1, do the following.

The integer value in doubleword element i of VSR[VRA+32] is multiplied by the integer value in doubleword element i of VSR[VRB+32].

The low-order 64 bits of the product are placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vmulld

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
</tbody>
</table>

0 64 127
### 6.9.1.4 Vector Integer Multiply-Add/Sum Instructions

#### Vector Multiply-High-Add Signed Halfword Saturate VA-form

```plaintext
vmhaddshs VRT, VRA, VRB, VRC
```

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
<td>32</td>
</tr>
</tbody>
</table>

```plaintext
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
	src1 ← EXTS(VSR[VRA+32].hword[i])
	src2 ← EXTS(VSR[VRB+32].hword[i])
	src3 ← EXTS(VSR[VRC+32].hword[i])

result ← ((src1 × src2)) >> 15 + src3
VSR[VRT+32].hword[i] ← si16_CLAMP(result)
VSCR.SAT ← sat_flag
end
```

For each integer value \( i \) from 0 to 7, do the following.

The signed integer value in halfword element \( i \) of VSR[VRA+32] is multiplied by the signed integer value in halfword element \( i \) of VSR[VRB+32], producing a 32-bit signed integer product.

Bits 0:16 of the product are added to the signed integer value in halfword element \( i \) of VSR[VRC+32].

- If the intermediate result is greater than \( 2^{15} \), the result saturates to \( 2^{15} - 1 \) and SAT is set to 1.
- If the intermediate result is less than \( -2^{15} \), the result saturates to \( -2^{15} \) and SAT is set to 1.

The low-order 16 bits of the result are placed into halfword element \( i \) of VSR[VRT+32].

Special Registers Altered:

- SAT

#### Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form

```plaintext
vmhraddshs VRT, VRA, VRB, VRC
```

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
<td>33</td>
</tr>
</tbody>
</table>

```plaintext
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
	src1 ← EXTS(VSR[VRA+32].hword[i])
	src2 ← EXTS(VSR[VRB+32].hword[i])
	src3 ← EXTS(VSR[VRC+32].hword[i])

result ← (((src1 × src2) + 0x0000_4000) >> 15) + src3
VSR[VRT+32].hword[i] ← si16_CLAMP(result)
VSCR.SAT ← sat_flag
end
```

For each integer value \( i \) from 0 to 7, do the following.

The signed integer value in halfword element \( i \) of VSR[VRA+32] is multiplied by the signed integer value in halfword element \( i \) of VSR[VRB+32], producing a 32-bit signed integer product.

The value 0x0000_4000 is added to the product.

Bits 0:16 of the 32-bit sum are added to the signed integer value in halfword element \( i \) of VSR[VRT+32].

- If the intermediate result is greater than \( 2^{15} - 1 \), the result saturates to \( 2^{15} - 1 \) and SAT is set to 1.
- If the intermediate result is less than \( -2^{15} \), the result saturates to \( -2^{15} \) and SAT is set to 1.

The low-order 16 bits of the result are placed into halfword element \( i \) of VSR[VRT+32].

Special Registers Altered:

- SAT

#### Register Data Layout for vmhaddshs & vmhraddshs

|------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|
Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form

\[ \text{VMLADDUH} \text{M VRT,VRA,VRB,VRC} \]

- If MSR.VEC=0 then Vector_Unavailable()
- do \( i = 0 \) to 7
  - src1 ← EXTZ(VSR[VRA+32].hword[i])
  - src2 ← EXTZ(VSR[VRB+32].hword[i])
  - src3 ← EXTZ(VSR[VRC+32].hword[i])
  - VSR[VRT+32].hword[i] ← \( \text{CHOP16}(\text{src1} \times \text{src2}) + \text{src3} \)
- end

For each integer value \( i \) from 0 to 7, do the following.

The unsigned integer value in halfword element \( i \) of VSR[VRA+32] is multiplied by the unsigned integer value in halfword element \( i \) of VSR[VRB+32].

The product is added to the unsigned integer value in halfword element \( i \) of VSR[VRC+32].

The low-order 16 bits of the sum of the product and the unsigned integer value in word element \( i \) of VSR[VRC+32] are placed into word element \( i \) of VSR[VRT+32].

Special Registers Altered:
None

Programming Note

\text{VMLADDUH} can be used for unsigned or signed-integers.

Register Data Layout for vmladduhm

|-------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|

Vector Multiply-Sum Unsigned Byte Modulo VA-form

\[ \text{VMSUMU} \text{BM VRT,VRA,VRB,VRC} \]

- If MSR.VEC=0 then Vector_Unavailable()
- do \( i = 0 \) to 3
  - temp ← EXTZ(VSR[VRC+32].word[i])
  - do \( j = 0 \) to 3
    - src1 ← EXTZ(VSR[VRA+32].word[i].byte[j])
    - src2 ← EXTZ(VSR[VRB+32].word[i].byte[j])
    - temp ← temp + (src1 \times src2)
  - end
  - VSR[VRT+32].word[i] ← \( \text{CHOP32}(\text{temp}) \)
- end

For each integer value \( i \) from 0 to 3, do the following.

For each integer value \( j \) from 0 to 3, do the following.

The unsigned integer value in byte element \( j \) of word element \( i \) of VSR[VRA+32] is multiplied by the unsigned integer value in byte element \( j \) of word element \( i \) of VSR[VRB+32].

The sum of the four products is added to the unsigned integer value in word element \( i \) of VSR[VRC+32].

The low-order 32 bits of the result are placed into word element \( i \) of VSR[VRT+32].

Special Registers Altered:
None

Programming Note

\text{VMSUMU} can be used for unsigned or signed-integers.

Register Data Layout for vmsumubm

|-------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
Vector Multiply-Sum Mixed Byte Modulo
VA-form

\[ \text{vmsummbm VRT,VRA,VRB,VRC} \]

\[
\text{if MSR.VEC=0 then Vector_Unavailable()}
\]

\[
\text{do i = 0 to 3}
\]

\[
\text{temp} \leftarrow \text{EXTS(VSR[VRC+32].word[i])}
\]

\[
\text{do j = 0 to 3}
\]

\[
\text{src1} \leftarrow \text{EXTS(VSR[VRA+32].word[i].byte[j])}
\]

\[
\text{src2} \leftarrow \text{EXTZ(VSR[VRB+32].word[i].byte[j])}
\]

\[
\text{temp} \leftarrow \text{temp} + (\text{src1} \times \text{src2})
\]

\[
\text{VSR[VRT+32].word[i]} \leftarrow \text{CHOP32(temp)}
\]

For each integer value \(i\) from 0 to 3, do the following.

For each integer value \(j\) from 0 to 3, do the following.

The signed integer value in byte element \(j\) of word element \(i\) of VSR[VRA+32] is multiplied by the unsigned integer value in byte element \(j\) of word element \(i\) of VSR[VRB+32].

The sum of the four products is added to the signed integer value in word element \(i\) of VSR[VRC+32].

The low-order 32 bits of the result are placed into word element \(i\) of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vmsummbm

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[0]</td>
<td>byte[1]</td>
<td>VSR[VRC+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
</tbody>
</table>

Vector Multiply-Sum Signed Halfword Modulo
VA-form

\[ \text{vmsumshm VRT,VRA,VRB,VRC} \]

\[
\text{if MSR.VEC=0 then Vector_Unavailable()}
\]

\[
\text{do i = 0 to 3}
\]

\[
\text{temp} \leftarrow \text{EXTS(VSR[VRC+32].word[i])}
\]

\[
\text{do j = 0 to 1}
\]

\[
\text{src1} \leftarrow \text{EXTS(VSR[VRA+32].hword[j])}
\]

\[
\text{src2} \leftarrow \text{EXTS(VSR[VRB+32].hword[j])}
\]

\[
\text{temp} \leftarrow \text{temp} + (\text{src1} \times \text{src2})
\]

\[
\text{VSR[VRT+32].word[i]} \leftarrow \text{CHOP32(temp)}
\]

For each integer value \(i\) from 0 to 3, do the following.

For each integer value \(j\) from 0 to 1, do the following.

The signed integer value in halfword element \(j\) of word element \(i\) of VSR[VRA+32] is multiplied by the signed integer value in halfword element \(j\) of word element \(i\) of VSR[VRB+32].

The sum of the two products is added to the signed integer value in word element \(i\) of VSR[VRC+32].

The low-order 32 bits of the result are placed into word element \(i\) of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vmsumshm

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>result</th>
</tr>
</thead>
</table>

Chapter 6. Vector Facility
**Vector Multiply-Sum Signed Halfword**

*Saturate VA-form*

\[ \text{vmsumshs } \text{VRT}, \text{VRA}, \text{VRB}, \text{VRC} \]

if MSR.VEC=0 then Vector_Unavailable()

\[
\text{do } i = 0 \text{ to } 3 \\
\text{  temp } \leftarrow \text{EXTS(VSR}[\text{VRC+32}].\text{word}[i]) \\
\text{  do } j = 0 \text{ to } 1 \\
\text{    src1 } \leftarrow \text{EXTS(VSR}[\text{VRA+32}].\text{word}[i].\text{hword}[j]) \\
\text{    src2 } \leftarrow \text{EXTS(VSR}[\text{VRB+32}].\text{word}[i].\text{hword}[j]) \\
\text{    temp } \leftarrow \text{temp } + (\text{src1 } \times \text{src2}) \\
\text{  end} \\
\text{VSR}[\text{VRT+32}].\text{word}[i] \leftarrow \text{si32_CLAMP(temp)} \\
\text{VSR}.\text{SAT} \leftarrow \text{sat\_flag} \\
\text{end}
\]

**For each integer value** \(i\) **from 0 to 3, do the following.**

**For each integer value** \(j\) **from 0 to 1, do the following.**

The signed integer value in halfword element \(j\) of word element \(i\) of VSR[VRT+32] is multiplied by the signed integer value in halfword element \(j\) of word element \(i\) of VSR[VRB+32].

The sum of the two products is added to the signed integer value in word element \(i\) of VSR[VRT+32].

- If the intermediate result is greater than \(2^{31}\), it saturates to \(2^{31}-1\) and SAT is set to 1,

- If the intermediate result is less than \(-2^{31}\) it saturates to \(-2^{31}\) and SAT is set to 1,

The low-order 32 bits of the result are placed into word element \(i\) of VSR[VRT+32].

**Special Registers Altered:**

\[\text{SAT}\]

**Register Data Layout for vmsumshs & vmsumumh**

|------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|

**Vector Multiply-Sum Unsigned Halfword**

*Modulo VA-form*

\[ \text{vmsumuhm } \text{VRT}, \text{VRA}, \text{VRB}, \text{VRC} \]

if MSR.VEC=0 then Vector_Unavailable()

\[
\text{do } i = 0 \text{ to } 3 \\
\text{  temp } \leftarrow \text{EXTZ(VSR}[\text{VRC+32}].\text{word}[i]) \\
\text{  do } j = 0 \text{ to } 1 \\
\text{    src1 } \leftarrow \text{EXTZ(VSR}[\text{VRA+32}].\text{word}[i].\text{hword}[j]) \\
\text{    src2 } \leftarrow \text{EXTZ(VSR}[\text{VRB+32}].\text{word}[i].\text{hword}[j]) \\
\text{    temp } \leftarrow \text{temp } + (\text{src1 } \times \text{src2}) \\
\text{  end} \\
\text{VSR}[\text{VRT+32}].\text{word}[i] \leftarrow \text{CHOP32(temp)} \\
\text{end}
\]

**For each integer value** \(i\) **from 0 to 3, do the following.**

**For each integer value** \(j\) **from 0 to 1, do the following.**

The unsigned integer value in halfword element \(j\) of word element \(i\) of VSR[VRT+32] is multiplied by the unsigned integer value in halfword element \(j\) of word element \(i\) of VSR[VRB+32].

The sum of the two products is added to the signed integer value in word element \(i\) of VSR[VRT+32].

The low-order 32 bits of the result are placed into word element \(i\) of VSR[VRT+32].

**Special Registers Altered:**

None

---

Version 3.1
Vector Multiply-Sum Unsigned Halfword
Saturate VA-form

\[ \text{vmsumuh} \quad \text{VRT, VRA, VRB, VRC} \]

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{i}</td>
<td>\text{6}</td>
<td>\text{VRT}</td>
<td>\text{VRA}</td>
<td>\text{VRB}</td>
<td>\text{VRC}</td>
<td>\text{39}</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do \text{i} = 0 to 3
\begin{align*}
\text{temp} & \leftarrow \text{EXTZ(VSR[VRC+32].word[i])} \\
\text{do \text{j} = 0 to 1}
\text{src1} & \leftarrow \text{EXTZ(VSR[VRA+32].word[i].hword[j])} \\
\text{src2} & \leftarrow \text{EXTZ(VSR[VRB+32].word[i].hword[j])} \\
\text{temp} & \leftarrow \text{temp + src1 \times src2} \\
\text{end}
\end{align*}
\text{VSR[VRT+32].word[i]} \leftarrow \text{ui32\_CLAMP(temp)}
\text{VSCR.SAT} \leftarrow \text{sat\_flag}
\text{end}

For each integer value \text{i} from 0 to 3, do the following.

For each integer value \text{j} from 0 to 1, do the following.

The unsigned integer value in halfword element \text{j} of word element \text{i} of VSR[VRA+32] is multiplied by the unsigned integer value in halfword element \text{j} of word element \text{i} of VSR[VRB+32].

The sum of the two products is added to the signed integer value in word element \text{i} of VSR[VRC+32].

- If the intermediate result is greater than \(2^{32}-1\), the result saturates to \(2^{32}-1\) and SAT is set to 1,
- If the intermediate result is less than \(-2^{32}\) it saturates to \(-2^{32}\) and SAT is set to 1,

The low-order 32 bits of the result are placed into word element \text{i} of VSR[VRT+32].

Special Registers Altered:
\text{SAT}

Register Data Layout for vmsumuh

|-----|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|
Vector Multiply-Sum Unsigned Doubleword Modulo VA-form

\[ \text{vmsumudm} \quad \text{VRT}, \text{VRA}, \text{VRB}, \text{VRC} \]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>VRC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[
\text{temp} \leftarrow \text{EXTZ(VSR}[\text{VRC+32}])
\]

do i = 0 to 1

\[
\text{src1} \leftarrow \text{EXTZ(VSR}[\text{VRA+32}].\text{dword}[i])
\]
\[
\text{src2} \leftarrow \text{EXTZ(VSR}[\text{VRB+32}].\text{dword}[i])
\]

\[
\text{temp} \leftarrow \text{temp} + (\text{src1} \times \text{src2})
\]
end

\[
\text{VSR}[\text{VRT+32}] \leftarrow \text{CHOP128}(\text{temp})
\]

Let \( \text{prod0} \) be the product of the unsigned integer values in doubleword element 0 of \( \text{VSR}[\text{VRA+32}] \) and doubleword element 0 of \( \text{VSR}[\text{VRB+32}] \).

Let \( \text{prod1} \) be the product of the unsigned integer values in doubleword element 1 of \( \text{VSR}[\text{VRA+32}] \) and doubleword element 1 of \( \text{VSR}[\text{VRB+32}] \).

The low-order 128 bits of the sum of \( \text{prod0}, \text{prod1}, \text{and the unsigned integer value in} \text{VSR}[\text{VRC+32}] \) are placed into \( \text{VSR}[\text{VRT+32}] \).

Special Registers Altered:
None

Register Data Layout for vmsumudm

| src1 | VSR[VA+32].dword[0] | VSR[VRB+32].dword[0] |
| src2 | VSR[VB+32].dword[0] | VSR[VRB+32].dword[0] |
| src3 | VSR[VC+32].dword[0] | VSR[VRB+32].dword[0] |
| result | VSR[VT+32] | VSR[VRB+32].dword[0] |

Programming Note

A horizontal add of the doubleword elements in \( \text{VSR}[\text{VRA+32}] \) can be performed using \text{vmsumudm} when \( \text{VSR}[\text{VRB+32}] \) contains the doubleword integer values \( \{1, 1\} \) and \( \text{VSR}[\text{VRC+32}] \) contains the quadword integer value 0.

A horizontal subtract of the doubleword elements in \( \text{VSR}[\text{VRA+32}] \) can be performed using \text{vmsumudm} when \( \text{VSR}[\text{VRB+32}] \) contains the doubleword integer values \( \{1, -1\} \) and \( \text{VSR}[\text{VRC+32}] \) contains the quadword integer value 0.

A multiply even unsigned doubleword operation can be performed using \text{vmsumudm} when the contents of doubleword element 0 of \( \text{VSR}[\text{VRA+32}] \) or \( \text{VSR}[\text{VRB+32}] \) are 0 and the contents of \( \text{VSR}[\text{VRC+32}] \) to 0.

A multiply odd unsigned doubleword operation can be performed using \text{vmsumudm} when the contents of doubleword element 0 of \( \text{VSR}[\text{VRA+32}] \) or \( \text{VSR}[\text{VRB+32}] \) are 0 and the contents of \( \text{VSR}[\text{VRC+32}] \) to 0.
Vector Multiply-Sum & write Carry-out
Unsigned Doubleword VA-form

\[ \text{vmsumcud} \quad \text{VRT,VRA,VRB,VRC} \]

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>VRT</th>
<th>11</th>
<th>VRA</th>
<th>16</th>
<th>VRB</th>
<th>21</th>
<th>VRC</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

temp ← EXTZ(VSR[VRC+32])

do i = 0 to 1

src1 ← EXTZ(VSR[VRA+32].dword[i])

src2 ← EXTZ(VSR[VRB+32].dword[i])

temp ← temp + (src1 × src2)
end

VSR[VRT+32] ← CHOP128(temp >> 128)

Let \( \text{prod0} \) be the quadword product of the unsigned integer values in doubleword element 0 of VSR[VRA+32] and doubleword element 0 of VSR[VRB+32].

Let \( \text{prod1} \) be the quadword product of the unsigned integer values in doubleword element 1 of VSR[VRA+32] and doubleword element 1 of VSR[VRB+32].

The carry out of the low-order 128 bits of the sum of \( \text{prod0} \), \( \text{prod1} \), and the unsigned integer value in VSR[VRC+32] is placed into VSR[VRT+32].

**Special Registers Altered:**

None

**Register Data Layout for vmsumcud**

\[
\begin{align*}
\text{src1} & : & \text{VSR[VRA+32].dword[0]} & : & \text{VSR[VRA+32].dword[1]} \\
\text{src2} & : & \text{VSR[VRB+32].dword[0]} & : & \text{VSR[VRB+32].dword[1]} \\
\text{src3} & : & \text{VSR[VRC+32].dword[0]} & : & \text{VSR[VRC+32].dword[1]} \\
\text{result} & : & \text{VSR[VRT+32]}
\end{align*}
\]
### Vector Integer Divide Instructions

#### Vector Divide Signed Word VX-form

**vdivsw**  
**VRT, VRA, VRB**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>395</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 3 \)
  - \( \text{dividend} \leftarrow \text{EXTS(VSR[VRA+32].word}[i]) \)
  - \( \text{divisor} \leftarrow \text{EXTS(VSR[VRB+32].word}[i]) \)
  - \( \text{VSR[VRT+32].word}[i] \leftarrow \text{CHOP32(dividend ÷ divisor)} \)

For each integer value \( i \) from 0 to 3, do the following.

- The signed integer in word element \( i \) of VSR[VRA+32] is divided by the signed integer in word element \( i \) of VSR[VRB+32].
- The quotient is the unique signed integer that satisfies
  \[
  \text{dividend} = (\text{quotient } \times \text{divisor}) + \text{remainder}
  \]
  where \( 0 \leq r < |\text{divisor}| \) if the dividend is nonnegative, and \( -|\text{divisor}| < r < 0 \) if the dividend is negative.
- If an attempt is made to perform any of the divisions
  - \( 0x8000_0000 \div -1 \)
  - \(<\text{anything}> + 0 \)
  then the quotient is undefined.
- The quotient is placed into word element \( i \) of VSR[VRT+32]

**Special Registers Altered:**  
None

**Register Data Layout for vdivsw & vdivuw**

|------|---------------------|---------------------|---------------------|---------------------|

#### Vector Divide Unsigned Word VX-form

**vdivuw**  
**VRT, VRA, VRB**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>139</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 3 \)
  - \( \text{dividend} \leftarrow \text{EXTZ(VSR[VRA+32].word}[i]) \)
  - \( \text{divisor} \leftarrow \text{EXTZ(VSR[VRB+32].word}[i]) \)
  - \( \text{VSR[VRT+32].word}[i] \leftarrow \text{CHOP32(dividend ÷ divisor)} \)

For each integer value \( i \) from 0 to 3, do the following.

- The unsigned integer in word element \( i \) of VSR[VRA+32] is divided by the unsigned integer in word element \( i \) of VSR[VRB+32].
- The quotient is the unique unsigned integer that satisfies
  \[
  \text{dividend} = (\text{quotient } \times \text{divisor}) + \text{remainder}
  \]
  where \( 0 \leq r < |\text{divisor}| \).
- If an attempt is made to perform the division
  - \(<\text{anything}> + 0 \)
  then the quotient is undefined.
- The quotient is placed into word element \( i \) of VSR[VRT+32]

**Special Registers Altered:**  
None
Vector Divide Extended Signed Word VX-form

vdivesw  VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>907</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   dividend ← EK5(VSR[VRA+32].word[i]) << 32
   divisor ← EK5(VSR[VRB+32].word[i])
   VSR[VRT+32].word[i] ← CHOP32(dividend ÷ divisor)
end

For each integer value i from 0 to 3, do the following.

Let dividend be the signed integer value in word element i of VSR[VRA+32], shifted left by 32 bits.
Let divisor be the signed integer value in word element i of VSR[VRB+32].
dividend is divided by divisor.
The quotient is the unique signed integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}
\]

where 0 ≤ r < |divisor| if the dividend is nonnegative, and -|divisor| < r ≤ 0 if the dividend is negative.

If the quotient cannot be represented in 32 bits, or if an attempt is made to perform the division,

<anything> + 0

the quotient is undefined.
The quotient is placed into word element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vdivesw & vdiveuw

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
</table>

Vector Divide Extended Unsigned Word VX-form

vdiveuw  VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>651</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   dividend ← EK2Z(VSR[VRA+32].word[i]) << 32
   divisor ← EK2Z(VSR[VRB+32].word[i])
   VSR[VRT+32].word[i] ← CHOP32(dividend ÷ divisor)
end

For each integer value i from 0 to 3, do the following.

Let dividend be the unsigned integer value in word element i of VSR[VRA+32], shifted left by 32 bits.
Let divisor be the unsigned integer value in word element i of VSR[VRB+32].
The quotient is the unique unsigned integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}
\]

where 0 ≤ r < |divisor|.

If the quotient cannot be represented in 32 bits, or if an attempt is made to perform the division,

<anything> + 0

the quotient is undefined.
The quotient is placed into word element i of VSR[VRT+32]

Special Registers Altered:
None
Vector Divide Signed Doubleword VX-form

\[
vdivsd \text{ VRT, VRA, VRB}
\]

if MSR.VEC=0 then Vector_Unavailable()

\[
do \ i = 0 \ to \ 1 \\
\begin{align*}
\text{dividend} & \leftarrow \text{EXTS}(\text{VSR}[\text{VRA+32}].dword[i]) \\
\text{divisor} & \leftarrow \text{EXTS}(\text{VSR}[\text{VRB+32}].dword[i]) \\
\text{VSR}[\text{VRT+32}].dword[i] & \leftarrow \text{CHOP64}(\text{dividend} \div \text{divisor})
\end{align*}
end
\]

For each integer value \(i\) from 0 to 1, do the following.

The signed integer in doubleword element \(i\) of VSR[VRT+32] is divided by the signed integer in doubleword element \(i\) of VSR[VRB+32].

The quotient is the unique signed integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}
\]

where \(0 \leq \text{remainder} < |\text{divisor}|\) if the dividend is nonnegative, and \(|\text{divisor}| < \text{remainder} \leq 0\) if the dividend is negative.

If an attempt is made to perform any of the divisions

\[
0x8000_0000_0000_0000 \div -1 \\
<\text{anything}> \div 0
\]

then the quotient is undefined.

The quotient is placed into word element \(i\) of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vdivsd & vdivud

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRT+32].dword[0]</th>
<th>VSR[VRT+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
</tbody>
</table>

Vector Divide Unsigned Doubleword VX-form

\[
vdivud \text{ VRT, VRA, VRB}
\]

if MSR.VEC=0 then Vector_Unavailable()

\[
do \ i = 0 \ to \ 1 \\
\begin{align*}
\text{dividend} & \leftarrow \text{EXTZ}(\text{VSR}[\text{VRA+32}].dword[i]) \\
\text{divisor} & \leftarrow \text{EXTZ}(\text{VSR}[\text{VRB+32}].dword[i]) \\
\text{VSR}[\text{VRT+32}].dword[i] & \leftarrow \text{CHOP64}(\text{dividend} \div \text{divisor})
\end{align*}
end
\]

For each integer value \(i\) from 0 to 1, do the following.

The unsigned integer in doubleword element \(i\) of VSR[VRT+32] is divided by the unsigned integer in doubleword element \(i\) of VSR[VRB+32].

The quotient is the unique unsigned integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}
\]

where \(0 \leq \text{remainder} < \text{divisor}\).

If an attempt is made to perform the division

\[
<\text{anything}> \div 0
\]

then the quotient is undefined.

The quotient is placed into doubleword element \(i\) of VSR[VRT+32].

Special Registers Altered:

None
### Vector Divide Extended Signed Doubleword VX-form

**vdivesd**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>971</td>
<td>971</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 1 \)
- \( \text{dividend} \leftarrow \text{EXTS}(\text{VSR[VRA+32].dword}[i]) \ll 64 \)
- \( \text{divisor} \leftarrow \text{EXTS}(\text{VSR[VRB+32].dword}[i]) \)
- \( \text{VSR[VRT+32].dword}[i] \leftarrow \text{CHOP64}(\text{dividend} \div \text{divisor}) \)

For each integer value \( i \) from 0 to 1, do the following.

Let \( \text{dividend} \) be the signed integer value in doubleword element \( i \) of VSR[VRA+32], shifted left by 64 bits.

Let \( \text{divisor} \) be the signed integer value in doubleword element \( i \) of VSR[VRB+32].

\( \text{dividend} \) is divided by \( \text{divisor} \).

The quotient is the unique signed integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}
\]

where \( 0 \leq r < |\text{divisor}| \) if the dividend is nonnegative, and \( -|\text{divisor}| < r \leq 0 \) if the dividend is negative.

If the quotient cannot be represented in 64 bits, or if an attempt is made to perform the division, \(<\text{anything}> + 0\), the quotient is undefined.

The quotient is placed into doubleword element \( i \) of VSR[VRT+32].

**Special Registers Altered:**

None

**Register Data Layout for vdivesd & vdiveud**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRA+32].dword[0]</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>

### Vector Divide Extended Unsigned Doubleword VX-form

**vdiveud**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>715</td>
<td>715</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 1 \)
- \( \text{dividend} \leftarrow \text{EXTZ}(\text{VSR[VRA+32].dword}[i]) \ll 64 \)
- \( \text{divisor} \leftarrow \text{EXTZ}(\text{VSR[VRB+32].dword}[i]) \)
- \( \text{VSR[VRT+32].dword}[i] \leftarrow \text{CHOP64}(\text{dividend} \div \text{divisor}) \)

For each integer value \( i \) from 0 to 1, do the following.

Let \( \text{dividend} \) be the unsigned integer value in doubleword element \( i \) of VSR[VRA+32], shifted left by 64 bits.

Let \( \text{divisor} \) be the unsigned integer value in doubleword element \( i \) of VSR[VRB+32].

The quotient is the unique unsigned integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}
\]

where \( 0 \leq r < |\text{divisor}| \).

If the quotient cannot be represented in 64 bits, or if an attempt is made to perform the division, \(<\text{anything}> + 0\), the quotient is undefined.

The quotient is placed into doubleword element \( i \) of VSR[VRT+32].

**Special Registers Altered:**

None
Vector Divide Signed Quadword VX-form

\[ \text{vdivsq} \quad \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>267</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector-Unavailable()
- \[ \text{dividend} \leftarrow \text{EXTS(VSR}[\text{VRA+32}]) \]
- \[ \text{divisor} \leftarrow \text{EXTS(VSR}[\text{VRB+32}]) \]
- \[ \text{VSR}[\text{VRT+32}] \leftarrow \text{CHOP128(dividend + divisor)} \]

Let \( \text{src1} \) be the signed integer value in \( \text{VSR}[\text{VRA+32}] \).
Let \( \text{src2} \) be the signed integer value in \( \text{VSR}[\text{VRB+32}] \).

The quotient of \( \text{src1} \) divided by \( \text{src2} \) is placed into \( \text{VSR}[\text{VRT+32}] \).

The quotient is the unique signed integer that satisfies

\[ \text{dividend} = (\text{quotient} \times \text{divisor}) + r \]

where \( 0 \leq r < |\text{divisor}| \) if the dividend is nonnegative, and \( -|\text{divisor}| < r \leq 0 \) if the dividend is negative.

If an attempt is made to perform any of the divisions

\[ 0\times8000_0000_0000_0000 \div -1 \]
\[ <\text{anything}> \div 0 \]

then the contents of \( \text{VSR}[\text{VRT+32}] \) are undefined.

Special Registers Altered:
None

Register Data Layout for \text{vdivs}q & \text{vdivu}q

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

Vector Divide Unsigned Quadword VX-form

\[ \text{vdivuq} \quad \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector-Unavailable()
- \[ \text{dividend} \leftarrow \text{EXTZ(VSR}[\text{VRA+32}]) \]
- \[ \text{divisor} \leftarrow \text{EXTZ(VSR}[\text{VRB+32}]) \]
- \[ \text{VSR}[\text{VRT+32}] \leftarrow \text{CHOP128(dividend + divisor)} \]

Let \( \text{src1} \) be the unsigned integer value in \( \text{VSR}[\text{VRA+32}] \).
Let \( \text{src2} \) be the unsigned integer value in \( \text{VSR}[\text{VRB+32}] \).

The quotient of \( \text{src1} \) divided by \( \text{src2} \) is placed into \( \text{VSR}[\text{VRT+32}] \).

The quotient is the unique unsigned integer that satisfies

\[ \text{dividend} = (\text{quotient} \times \text{divisor}) + r \]

where \( 0 \leq r < \text{divisor} \).

If an attempt is made to perform the division

\[ <\text{anything}> \div 0 \]

then the contents of \( \text{VSR}[\text{VRT+32}] \) are undefined.

Special Registers Altered:
None
Vector Divide Extended Signed Quadword VX-form

\[
vxdividesq \quad VRT, VRA, VRB
\]

\[
\begin{array}{c|cccc|c}
4 & 6 & 11 & 16 & 21 & 779
\end{array}
\]

\[\text{if MSR.VEC}=0 \text{ then Vector.Unavailable();}\]

\[\text{dividend} \leftarrow \text{EXTS(VSR[VRA+32])} \ll 128\]
\[\text{divisor} \leftarrow \text{EXTS(VSR[VRB+32])}\]

\[\text{VSR[VRT+32]} \leftarrow \text{CHOP128(dividend + divisor)}\]

Let \(src1\) be the signed integer value in VSR[VRA+32] concatenated with 128 0s.

Let \(src2\) be the signed integer value in VSR[VRB+32].

The quotient of \(src1\) divided by \(src2\) is placed into VSR[VRT+32].

The quotient is the unique signed integer that satisfies

\[\text{dividend} = (\text{quotient} \times \text{divisor}) + r\]

where \(0 \leq r < |\text{divisor}|\) if dividend is nonnegative, and \(-|\text{divisor}| < r \leq 0\) if dividend is negative.

If the quotient cannot be represented in 128 bits, or if an attempt is made to perform the division

\[<\text{anything}> + 0\]

then the contents of VSR[VRT+32] are undefined.

Special Registers Altered:

None

Vector Divide Extended Unsigned Quadword VX-form

\[
vxdivieuq \quad VRT, VRA, VRB
\]

\[
\begin{array}{c|cccc|c}
4 & 6 & 11 & 16 & 21 & 523
\end{array}
\]

\[\text{if MSR.VEC}=0 \text{ then Vector.Unavailable();}\]

\[\text{dividend} \leftarrow \text{EXTZ(VSR[VRA+32])} \ll 128\]
\[\text{divisor} \leftarrow \text{EXTZ(VSR[VRB+32])}\]

\[\text{VSR[VRT+32]} \leftarrow \text{CHOP128(dividend + divisor)}\]

Let \(src1\) be the unsigned integer value in VSR[VRA+32] concatenated with 128 0s.

Let \(src2\) be the unsigned integer value in VSR[VRB+32].

The quotient of \(src1\) divided by \(src2\) is placed into VSR[VRT+32].

The quotient is the unique unsigned integer that satisfies

\[\text{dividend} = (\text{quotient} \times \text{divisor}) + r\]

where \(0 \leq r < \text{divisor}\).

If the quotient cannot be represented in 128 bits, or if an attempt is made to perform the division

\[<\text{anything}> + 0\]

then the contents of VSR[VRT+32] are undefined.

Special Registers Altered:

None

Register Data Layout for vxdivesq & vxdiveuq

\[
\begin{array}{ccc}
\text{src1} & \text{VSR[VRA+32]} \\
\text{src2} & \text{VSR[VRB+32]} \\
\text{result} & \text{VSR[VRT+32]} \\
0 & 127
\end{array}
\]
6.9.1.6 Vector Integer Modulo Instructions

**Vector Modulo Signed Word VX-form**

vmodsw  VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1931</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3

\[
\text{dividend} \leftarrow \text{EXTS(VSR}[\text{VRA+32}].\text{word}[i])
\]

\[
\text{divisor} \leftarrow \text{EXTS(VSR}[\text{VRB+32}].\text{word}[i])
\]

\[
\text{VSR}[\text{VRT+32}].\text{word}[i] \leftarrow \text{CHOP32(dividend \% divisor)}
\]

end

For each integer value \( i \) from 0 to 3, do the following.

The signed integer in word element \( i \) of VSR[VRB+32] is divided by the signed integer in word element \( i \) of VSR[VRA+32].

The remainder is the unique signed integer that satisfies

\[
\text{quotient} = \text{dividend} / \text{divisor}
\]

\[
\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})
\]

where \( 0 \leq \text{remainder} < |\text{divisor}| \) if the dividend is nonnegative, and \( -|\text{divisor}| < \text{remainder} \leq 0 \) if the dividend is negative.

If an attempt is made to perform any of the modulo operations

\[
0 \times 0 \text{%}_0 \quad 0 \times 0 \text{%} -1
\]

then the remainder is undefined.

The remainder is placed into word element \( i \) of VSR[VRT+32]

**Special Registers Altered:**

None

---

**Vector Modulo Unsigned Word VX-form**

vmoduw  VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1675</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3

\[
\text{dividend} \leftarrow \text{EXTZ(VSR}[\text{VRA+32}].\text{word}[i])
\]

\[
\text{divisor} \leftarrow \text{EXTZ(VSR}[\text{VRB+32}].\text{word}[i])
\]

\[
\text{VSR}[\text{VRT+32}].\text{word}[i] \leftarrow \text{CHOP32(dividend \% divisor)}
\]

end

For each integer value \( i \) from 0 to 3, do the following.

The unsigned integer in word element \( i \) of VSR[VRB+32] is divided by the unsigned integer in word element \( i \) of VSR[VRA+32].

The remainder is the unique unsigned integer that satisfies

\[
\text{quotient} = \text{dividend} / \text{divisor}
\]

\[
\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})
\]

where \( 0 \leq \text{remainder} < \text{divisor} \).

If an attempt is made to perform the modulo operation

\[
<\text{anything}> \% 0
\]

then the remainder is undefined.

The remainder is placed into word element \( i \) of VSR[VRT+32]

**Special Registers Altered:**

None

---

**Register Data Layout for vmodsw & vmoduw**

|------|---------------------|---------------------|---------------------|---------------------|

0 32 64 96 127

---

364  Power ISA™ I
Vector Modulo Signed Doubleword VX-form

```plaintext
vmodsd VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 1
    dividend ← ETS(VSR[VRA+32].dword[i])
    divisor ← ETS(VSR[VRB+32].dword[i])
    VSR[VRT+32].dword[i] ← CHOP64(dividend % divisor)
end
```

For each integer value \( i \) from 0 to 1, do the following.

The signed integer in doubleword element \( i \) of VSR[VRA+32] is divided by the signed integer in doubleword element \( i \) of VSR[VRB+32].

The remainder is the unique signed integer that satisfies:

\[
\text{quotient} = \frac{\text{dividend}}{\text{divisor}} \\
\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})
\]

where \( 0 \leq \text{remainder} < |\text{divisor}| \) if the dividend is nonnegative, and \( |\text{divisor}| < \text{remainder} \leq 0 \) if the dividend is negative.

If an attempt is made to perform any of the modulo operations

\[
0x8000_0000 \ % -1 \\
<\text{anything}> \ % 0
\]

the remainder is undefined.

The remainder is placed into doubleword element \( i \) of VSR[VRT+32]

**Special Registers Altered:**

None

**Register Data Layout for vdivesd & vdiveud**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRA+32].dword[0]</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>

Vector Modulo Unsigned Doubleword VX-form

```plaintext
vmodud VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 1
    dividend ← ETSZ(VSR[VRA+32].dword[i])
    divisor ← ETSZ(VSR[VRB+32].dword[i])
    VSR[VRT+32].dword[i] ← CHOP64(dividend % divisor)
end
```

For each integer value \( i \) from 0 to 1, do the following.

The unsigned integer in doubleword element \( i \) of VSR[VRA+32] is divided by the unsigned integer in doubleword element \( i \) of VSR[VRB+32].

The remainder is the unique unsigned integer that satisfies:

\[
\text{quotient} = \frac{\text{dividend}}{\text{divisor}} \\
\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})
\]

where \( 0 \leq \text{remainder} < \text{divisor} \).

If an attempt is made to perform the modulo operation

\[
<\text{anything}> \ % 0
\]

the remainder is undefined.

The remainder is placed into doubleword element \( i \) of VSR[VRT+32]

**Special Registers Altered:**

None

**Register Data Layout for vdivesd & vdiveud**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRA+32].dword[0]</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>
Vector Modulo Signed Quadword VX-form  
\[ \vmodsq \text{ VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1803</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable() 

\[
\begin{align*}
\text{dividend} & \leftarrow \text{EXTS}(VSR[VRA+32]) \\
\text{divisor} & \leftarrow \text{EXTS}(VSR[VRB+32]) \\
VSR[VRT+32] & \leftarrow \text{CHOP128}(\text{dividend} \mod \text{divisor})
\end{align*}
\]

Let src1 be the signed integer value in VSR[VRA+32]. Let src2 be the signed integer value in VSR[VRB+32].

The remainder of src1 divided by src2 is placed into VSR[VRT+32].

The remainder is the unique signed integer that satisfies

\[
\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})
\]

where \(0 \leq \text{remainder} < |\text{divisor}|\) if the dividend is nonnegative, and \(-|\text{divisor}| < \text{remainder} \leq 0\) if the dividend is negative.

If an attempt is made to perform any of the divisions

\[
\langle \text{anything} \rangle \mod 0, 0x8000_0000_0000_0000 \mod -1
\]

then the contents of VSR[VRT+32] are undefined.

Special Registers Altered:

None

Register Data Layout for vmodsq & vmoduq

\[
\begin{align*}
\text{src1} & \rightarrow VSR[VRA+32] \\
\text{src2} & \rightarrow VSR[VRB+32] \\
\text{result} & \rightarrow VSR[VRT+32]
\end{align*}
\]

Vector Modulo Unsigned Quadword VX-form  
\[ \vmoduq \text{ VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1547</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable() 

\[
\begin{align*}
\text{dividend} & \leftarrow \text{EXTZ}(VSR[VRA+32]) \\
\text{divisor} & \leftarrow \text{EXTZ}(VSR[VRB+32]) \\
VSR[VRT+32] & \leftarrow \text{CHOP128}(\text{dividend} \mod \text{divisor})
\end{align*}
\]

Let src1 be the signed integer value in VSR[VRA+32]. Let src2 be the signed integer value in VSR[VRB+32].

The remainder of src1 divided by src2 is placed into VSR[VRT+32].

The remainder is the unique signed integer that satisfies

\[
\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})
\]

where \(0 \leq \text{remainder} < \text{divisor}\).

If an attempt is made to perform any of the divisions

\[
\langle \text{anything} \rangle \mod 0
\]

then the contents of VSR[VRT+32] are undefined.

Special Registers Altered:

None
6.9.1.7 Vector Integer Sum-Across Instructions

**Vector Sum across Signed Word Saturate VX-form**

```vsumsws VRT,VRA,VRB```

if MSR.VEC=0 then Vector_Unavailable();

```temp ← EXTS(VSR[VRA+32].word[3])```  
do i = 0 to 3
  ```temp ← temp + EXTS(VSR[VRA+32].word[i])```  
end

VSR[VRT+32].word[0] ← 0x0000_0000  
VSR[VRT+32].word[1] ← 0x0000_0000  
VSR[VRT+32].word[2] ← 0x0000_0000  
VSR[VRT+32].word[3] ← si32_CLAMP(temp)  
VSCR.SAT ← sat_flag

The sum of the signed integer values in the four word elements of `VSR[VRA+32]` is added to the signed integer value in the word element 3 of `VSR[VRB+32]`.

- If the intermediate result is greater than `2^{31}-1` the result saturates to `2^{31}-1` and SAT is set to 1.
- If the intermediate result is less than `-2^{31}` the result saturates to `-2^{31}` and SAT is set to 1.

The low-order 32 bits of the result are placed into word element 3 of `VSR[VRT+32]`.

Word elements 0 to 2 of `VSR[VRT+32]` are set to 0.

**Special Registers Altered:**

SAT

**Register Data Layout for vsumsws**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>0x0000_0000</td>
<td>0x0000_0000</td>
<td>0x0000_0000</td>
<td>VSR[VRT+32].word[3]</td>
</tr>
</tbody>
</table>
**Vector Sum across Half Signed Word Saturate VX-form**

\[ \text{vsum2sws VRT,VRA,VRB} \]

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>1672</td>
<td></td>
</tr>
</tbody>
</table>

\begin{align*}
\text{if MSR.VEC=0 then VectorUnavailable} & \end{align*}

\begin{align*}
do \ i \ = \ 0 \ \text{to} \ 1 \\
& \text{temp} \leftarrow \text{EXTS(VSR[VRB+32].dword[i].word[1])} \\
do \ j \ = \ 0 \ \text{to} \ 1 \\
& \text{temp} \leftarrow \text{temp} + \text{EXTS(VSR[VRA+32].dword[i].word[j])} \\
\end{align*}

\begin{align*}
\text{VSR[VRT+32].dword[i].word[0]} & \leftarrow 0x0000_0000 \\
\text{VSR[VRT+32].dword[i].word[1]} & \leftarrow \text{si32_CLAMP(temp)} \\
\text{VSCR.SAT} & \leftarrow \text{sat_flag} \\
\end{align*}

Word elements 0 and 2 of VSR[VRT+32] are set to 0.

The sum of the signed integer values in word elements 0 and 1 in VSR[VRA+32] is added to the signed integer value in word element 1 of VSR[VRB+32].

\begin{align*}
\text{if intermediate result is greater than } 2^{31} - 1 \text{ the } \\
\text{result saturates to } 2^{31} - 1 \text{ and SAT is set to 1.} \\
\text{if intermediate result is less than } -2^{31} \text{ the } \\
\text{result saturates to } -2^{31} \text{ and SAT is set to 1.} \\
\end{align*}

The low-order 32 bits of the result are placed into word element 1 of VSR[VRT+32].

The sum of the signed integer values in word elements 2 and 3 in VSR[VRA+32] is added to the signed integer value in word element 3 of VSR[VRB+32].

\begin{align*}
\text{if intermediate result is greater than } 2^{31} - 1 \text{ the } \\
\text{result saturates to } 2^{31} - 1 \text{ and SAT is set to 1.} \\
\text{if intermediate result is less than } -2^{31} \text{ the } \\
\text{result saturates to } -2^{31} \text{ and SAT is set to 1.} \\
\end{align*}

The low-order 32 bits of the result are placed into word element 3 of VSR[VRT+32].

**Special Registers Altered:**

SAT

**Register Data Layout for vsum2sws**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>0x0000_0000</td>
<td>VSR[VRT+32].word[1]</td>
<td>0x0000_0000</td>
<td>VSR[VRT+32].word[3]</td>
</tr>
</tbody>
</table>
Vector Sum across Quarter Signed Byte
Saturate VX-form

```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 3
    temp ← EXTS(VSR[VRB+32].word[i])
    for j = 0 to 3
        temp ← temp + EXTS(VSR[VRA+32].word[i].byte[j])
    end
    VSR[VRT+32].word[i] ← si32_CLAMP(temp)
end
```

For each integer value \(i\) from 0 to 3, do the following.

The sum of the signed integer values in the four byte elements contained in word element \(i\) of \(VSR[VRA+32]\) is added to the signed integer value in word element \(i\) of \(VSR[VRB+32]\).

- If the intermediate result is greater than \(2^{31}-1\) the result saturates to \(2^{31}-1\) and SAT is set to 1.
- If the intermediate result is less than \(-2^{31}\) the result saturates to \(-2^{31}\) and SAT is set to 1.

The result is placed into word element \(i\) of \(VSR[VRT+32]\).

Special Registers Altered:
SAT

### Special Registers Altered:
SAT

### Register Data Layout for vsum4sbs

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>by[e][3]</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
<tr>
<td>by[e][2]</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
<tr>
<td>by[e][1]</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
<tr>
<td>by[e][0]</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
</tbody>
</table>

### Register Data Layout for vsum4shs

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[VSR[VRA+32].word[0]]</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
<tr>
<td>[VSR[VRA+32].word[1]]</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
<tr>
<td>[VSR[VRA+32].word[2]]</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
<tr>
<td>[VSR[VRA+32].word[3]]</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
</tbody>
</table>
Vector Sum across Quarter Unsigned Byte
Saturate VX-form

\[ \text{vsum4ubs } \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>6</th>
<th>VRA</th>
<th>16</th>
<th>VRB</th>
<th>21</th>
<th>1544</th>
</tr>
</thead>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{do } & i = 0 \text{ to } 3 \\
\text{temp} & \leftarrow \text{EXTZ(VSR[VRB+32].word[i])} \\
\text{do } & j = 0 \text{ to } 3 \\
\text{temp} & \leftarrow \text{temp} + \text{EXTZ(VSR[VRT+32].word[i].byte[j])} \\
\text{end} \\
\text{VSR[VRT+32].word[i]} & \leftarrow \text{ui32.Clamp} \text{temp} \\
\text{VSCR.SAT} & \leftarrow \text{sat_flag}
\end{align*}
\]

For each integer value \(i\) from 0 to 3, do the following.

The sum of the unsigned integer values in the four byte elements contained in word element \(i\) of VSR[VRA+32] is added to the unsigned integer value in word element \(i\) of VSR[VRB+32].

- If the intermediate result is greater than \(2^{32}-1\) it saturates to \(2^{32}-1\) and SAT is set to 1.

The result is placed into word element \(i\) of VSR[VRT+32].

Special Registers Altered:

SAT

Register Data Layout for vsum4ubs

|------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
6.9.1.8 Vector Integer Negate Instructions

Vector Negate Word VX-form

vnegw VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1538</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable();

\[\text{do } i = 0 \text{ to } 3\]

\[\text{src } \leftarrow \text{EXTS(VSR[VRB+32].word[i])}\]

\[\text{VSR[VRT+32].word[i] } \leftarrow \text{CHOP32}(\neg \text{src} + 1)\]

end

For each integer value \(i\) from 0 to 3, do the following.
The sum of the one’s-complement of the signed integer in word element \(i\) of VSR[VRB+32] and 1 is placed into word element \(i\) of VSR[VRT+32].

Special Registers Altered:
None

Vector Negate Doubleword VX-form

vnegd VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1538</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable();

\[\text{do } i = 0 \text{ to } 1\]

\[\text{src } \leftarrow \text{EXTS(VSR[VRB+32].dword[i])}\]

\[\text{VSR[VRT+32].dword[i] } \leftarrow \text{CHOP64}(\neg \text{src} + 1)\]

end

For each integer value \(i\) from 0 to 1, do the following.
The sum of the one’s-complement of the signed integer in doubleword element \(i\) of VSR[VRB+32] and 1 is placed into doubleword element \(i\) of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vnegw

<table>
<thead>
<tr>
<th>src</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>.byte[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
</tbody>
</table>

 Register Data Layout for vnegd

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRA+32].dword[0]</td>
<td>VSR[VRA+32].dword[1]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>127</th>
</tr>
</thead>
</table>

Chapter 6. Vector Facility 371
6.9.1.9 Vector Extend Sign Instructions

**Vector Extend Sign Byte To Word VX-form**

`vextsb2w VRT,VRB`

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>16</th>
<th>VRB</th>
<th>1538</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

Do i = 0 to 3

src ← VSR[VRB+32].word[i].bit[24:31]

VSR[VRT+32].word[i] ← EXT32(src)

End

For each integer value i from 0 to 3, do the following.

The signed integer in bits 24:31 of word element i of VSR[VRB+32] is sign-extended and placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

---

**Vector Extend Sign Halfword To Word VX-form**

`vextsh2w VRT,VRB`

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>17</th>
<th>VRB</th>
<th>1538</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>17</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

Do i = 0 to 3

src ← VSR[VRB+32].word[i].bit[16:31]

VSR[VRT+32].word[i] ← EXT32(src)

End

For each integer value i from 0 to 3, do the following.

The signed integer in bits 16:31 of word element i of VSR[VRB+32] is sign-extended and placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

---

**Register Data Layout for vextsb2w**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24</td>
<td>32</td>
<td>56</td>
<td>64</td>
<td>88</td>
<td>96</td>
<td>120</td>
<td>127</td>
</tr>
</tbody>
</table>

**Register Data Layout for vextsh2w**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
<td>64</td>
<td>80</td>
<td>96</td>
<td>104</td>
<td>127</td>
</tr>
</tbody>
</table>
Vector Extend Sign Byte To Doubleword
VX-form
vextsb2d  VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>24</th>
<th>16</th>
<th>21</th>
<th>1538</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td>VRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();

for i = 0 to 1 do

src ← VSR[VRB+32].dword[i].bit[56:63]
VSR[VRT+32].dword[i] ← EXT564(src)
end

For each integer value i from 0 to 1, do the following.
The signed integer in bits 56:63 of doubleword element i of VSR[VRB+32] is sign-extended and placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vextsb2d

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector Extend Sign Halfword To Doubleword
VX-form

vextsh2d  VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>25</th>
<th>16</th>
<th>21</th>
<th>1538</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td>VRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();

for i = 0 to 1 do

src ← VSR[VRB+32].dword[i].bit[48:63]
VSR[VRT+32].dword[i] ← EXT564(src)
end

For each integer value i from 0 to 1, do the following.
The signed integer in bits 48:63 of doubleword element i of VSR[VRB+32] is sign-extended and placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vextsh2d

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector Extend Sign Word To Doubleword

VX-form

\texttt{vextsw2d VRT,VRB}

\begin{verbatim}
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
    src ← VSR[VRB+32].dword[i].bit[32:63]
    VSR[VRT+32].dword[i] ← EXT64(src)
end

For each integer value \(i\) from 0 to 1, do the following.
The signed integer in bits 32:63 of doubleword element \(i\) of \(VSR[VRB+32]\) is sign-extended and placed into doubleword element \(i\) of \(VSR[VRT+32]\).

Special Registers Altered:
None

\end{verbatim}

Register Data Layout for \texttt{vextsw2d}

\begin{tabular}{c|c|c|c|c|c|c}
src & unused & \(VSR[VRT+32].word[1]\) & unused & \(VSR[VRT+32].word[0]\) & result
\end{tabular}

\begin{tabular}{c|c|c|c|c|c|c}
0 & 6 & 11 & 16 & 21 & 31
end

Vector Extend Sign Doubleword to Quadword

VX-form

\texttt{vextsd2q VRT,VRB}

\begin{verbatim}
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← EXT128(VSR[VRB+32].bit[64:127])

The signed integer in bits 64:127 of \(VSR[VRB+32]\) is signed extended to 128 bits and placed into \(VSR[VRT+32]\).

Special Registers Altered:
None

\end{verbatim}

Register Data Layout for \texttt{vextsd2q}

\begin{tabular}{c|c|c|c|c|c|c}
src & unused & \(VSR[VRB+32].dword[1]\) & result
\end{tabular}

\begin{tabular}{c|c|c|c|c|c|c}
0 & 6 & 11 & 16 & 21 & 31
end
6.9.1.10 Vector Integer Average Instructions

**Vector Average Signed Byte VX-form**

\[ \text{vavgsb} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1282</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
\[ \text{src1} \leftarrow \text{EXTS(VSR[VRA+32].byte[i])} \]
\[ \text{src2} \leftarrow \text{EXTS(VSR[VRB+32].byte[i])} \]
\[ \text{VSR[VRT+32].byte[i]} \leftarrow \text{CHOP8((src1 + src2 + 1) >> 1)} \]
end

For each integer value \( i \) from 0 to 15, do the following.
Let \( \text{src1} \) be the signed integer value in byte element \( i \) of \( \text{VSR[VRA+32]} \).
Let \( \text{src2} \) be the signed integer value in byte element \( i \) of \( \text{VSR[VRB+32]} \).
\( \text{src1} \) is added to \( \text{src2} \).
The sum is incremented by 1 and then shifted right 1 bit.
The low-order 8 bits of the result are placed into byte element \( i \) of \( \text{VSR[VRT+32]} \).

**Special Registers Altered:**
None

**Register Data Layout for vavgsb & vavgub**

\( \text{src1} \)
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
\( \text{src2} \)
\( \text{result} \)


**Vector Average Unsigned Byte VX-form**

\[ \text{vavgub} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1026</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
\[ \text{src1} \leftarrow \text{EXTZ(VSR[VRA+32].byte[i])} \]
\[ \text{src2} \leftarrow \text{EXTZ(VSR[VRB+32].byte[i])} \]
\[ \text{VSR[VRT+32].byte[i]} \leftarrow \text{CHOP8((src1 + src2 + 1) >> 1)} \]
end

For each integer value \( i \) from 0 to 15, do the following.
Let \( \text{src1} \) be the unsigned integer value in byte element \( i \) of \( \text{VSR[VRA+32]} \).
Let \( \text{src2} \) be the unsigned integer value in byte element \( i \) of \( \text{VSR[VRB+32]} \).
\( \text{src1} \) is added to \( \text{src2} \).
The sum is incremented by 1 and then shifted right 1 bit.
The low-order 8 bits of the result are placed into byte element \( i \) of \( \text{VSR[VRT+32]} \).

**Special Registers Altered:**
None
Vector Average Signed Halfword VX-form

\[ \text{vavgsh} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
src1 ← EXTS(VSR[VRA+32].hword[i])
src2 ← EXTS(VSR[VRB+32].hword[i])
VSR[VRT+32].hword[i] ← CHOP16((src1 + src2 + 1) >> 1)
end
```

For each integer value \( i \) from 0 to 7, do the following.

Let \( \text{src1} \) be the signed integer value in halfword element \( i \) of \( \text{VSR[VRA+32]} \).

Let \( \text{src2} \) be the signed integer value in halfword element \( i \) of \( \text{VSR[VRB+32]} \).

\( \text{src1} \) is added to \( \text{src2} \).

The sum is incremented by 1 and then shifted right 1 bit.

The low-order 16 bits of the result are placed into halfword element \( i \) of \( \text{VSR[VRT+32]} \).

Special Registers Altered:
None

Register Data Layout for \text{vavgsh} \& \text{vavguh}

|------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|

Vector Average Unsigned Halfword VX-form

\[ \text{vavguh} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
src1 ← EXTS1(VSR[VRA+32].hword[i])
src2 ← EXTS1(VSR[VRB+32].hword[i])
VSR[VRT+32].hword[i] ← CHOP16((src1 + src2 + 1) >> 1)
end
```

For each integer value \( i \) from 0 to 7, do the following.

Let \( \text{src1} \) be the unsigned integer value in halfword element \( i \) of \( \text{VSR[VRA+32]} \).

Let \( \text{src2} \) be the unsigned integer value in halfword element \( i \) of \( \text{VSR[VRB+32]} \).

\( \text{src1} \) is added to \( \text{src2} \).

The sum is incremented by 1 and then shifted right 1 bit.

The low-order 16 bits of the result are placed into halfword element \( i \) of \( \text{VSR[VRT+32]} \).

Special Registers Altered:
None
Vector Average Signed Word VX-form

\textbf{vavgsw} \quad \text{VRT}, \text{VRA}, \text{VRB}

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1410</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable().

\text{do } i = 0 \text{ to } 3
\quad \text{src1} \leftarrow \text{EXTS(VSR[VRA+32].word[i])}
\quad \text{src2} \leftarrow \text{EXTS(VSR[VRB+32].word[i])}
\quad \text{VSR}[VRT+32].word[i] \leftarrow \text{Chop32}((\text{src1} + \text{src2} + 1) >> 1)
\text{end}

For each integer value \(i\) from 0 to 3, do the following.

Let \text{src1} be the signed integer value in word element \(i\) of VSR[VRA+32].

Let \text{src2} be the signed integer value in word element \(i\) of VSR[VRB+32].

\text{src1} is added to \text{src2}.

The sum is incremented by 1 and then shifted right 1 bit.

The low-order 32 bits of the result are placed into word element \(i\) of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for \text{vavgsw} & \text{vavguw}

|------|---------------------|---------------------|---------------------|---------------------|

Vector Average Unsigned Word VX-form

\textbf{vavguw} \quad \text{VRT}, \text{VRA}, \text{VRB}

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1154</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable().

\text{do } i = 0 \text{ to } 3
\quad \text{src1} \leftarrow \text{EXTZ(VSR[VRA+32].word[i])}
\quad \text{src2} \leftarrow \text{EXTZ(VSR[VRB+32].word[i])}
\quad \text{VSR}[VRT+32].word[i] \leftarrow \text{Chop32}((\text{src1} + \text{src2} + 1) >> 1)
\text{end}

For each integer value \(i\) from 0 to 3, do the following.

Let \text{src1} be the unsigned integer value in word element \(i\) of VSR[VRA+32].

Let \text{src2} be the unsigned integer value in word element \(i\) of VSR[VRB+32].

\text{src1} is added to \text{src2}.

The sum is incremented by 1 and then shifted right 1 bit.

The low-order 32 bits of the result are placed into word element \(i\) of VSR[VRT+32].

Special Registers Altered:

None
6.9.1.11 Vector Integer Absolute Difference Instructions

This section describes a set of instructions that return the absolute value of the difference of integer values.

### Vector Absolute Difference Unsigned Byte

**VX-form**

<table>
<thead>
<tr>
<th>Vabsdub</th>
<th>VRT, VRA, VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
src1 ← EXTZ(VSR[VRA+32].byte[i])
src2 ← EXTZ(VSR[VRB+32].byte[i])
if src1 > src2 then
VSR[VRT+32].byte[i] ← CHOP8(src1 + ¬src2 + 1)
else
VSR[VRT+32].byte[i] ← CHOP8(src2 + ¬src1 + 1)
end

For each integer value i from 0 to 15, do the following.
Let src1 be the unsigned integer value in byte element i of VSR[VRA+32].
Let src2 be the unsigned integer value in byte element i of VSR[VRB+32].
src1 is subtracted by src2.
The absolute value of the difference is placed into byte element i of VSR[VRT+32].

**Special Registers Altered:**
None

### Register Data Layout for vabsdub

|------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|

|------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|

### Vector Absolute Difference Unsigned Halfword VX-form

<table>
<thead>
<tr>
<th>vabsduh</th>
<th>VRT, VRA, VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
src1 ← EXTZ(VSR[VRA+32].hword[i])
src2 ← EXTZ(VSR[VRB+32].hword[i])
if src1 > src2 then
VSR[VRT+32].hword[i] ← CHOP16(src1 + ¬src2 + 1)
else
VSR[VRT+32].hword[i] ← CHOP16(src2 + ¬src1 + 1)
end

For each integer value i from 0 to 7, do the following.
Let src1 be the unsigned integer value in halfword element i of VSR[VRA+32].
Let src2 be the unsigned integer value in halfword element i of VSR[VRB+32].
src1 is subtracted by src2.
The absolute value of the difference is placed into halfword element i of VSR[VRT+32].

**Special Registers Altered:**
None

### Register Data Layout for vabsduh

|------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|
Vector Absolute Difference Unsigned Word
**VX-form**

**vabsduw**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1155</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3

src1 ← EXTZ(VSR[VRA+32].word[i])

src2 ← EXTZ(VSR[VRB+32].word[i])

if src1 > src2 then

VSR[VRT+32].word[i] ← CHOP32(src1 + ¬src2 + 1)

else

VSR[VRT+32].word[i] ← CHOP32(src2 + ¬src1 + 1)
end

For each integer value i from 0 to 3, do the following.

Let src1 be the unsigned integer value in word element i of VSR[VRA+32].

Let src2 be the unsigned integer value in word element i of VSR[VRB+32].

src1 is subtracted by src2.

The absolute value of the difference is placed into word element i of VSR[VRT+32].

**Special Registers Altered:**
None

**Register Data Layout for vabsduw**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
<td>127</td>
</tr>
</tbody>
</table>
6.9.2 Vector Integer Maximum/Minimum Instructions

6.9.2.1 Vector Integer Maximum Instructions

**Vector Maximum Signed Byte VX-form**

<table>
<thead>
<tr>
<th>vxmaxsb</th>
<th>VRT, VRA, VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>VRT</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

- if MSR.VEC = 0 then Vector_Unavailable()
- do i = 0 to 15
  - src1 ← VSR[VRA+32].byte[i]
  - src2 ← VSR[VRB+32].byte[i]
  - gt_flag ← EXTS(src1) > EXTS(src2)
  - VSR[VRT+32].byte[i] ← gt_flag=1 ? src1 : src2
- end

For each integer value i from 0 to 15, do the following.
- Let src1 be the signed integer value in byte element i of VSR[VRA+32].
- Let src2 be the signed integer value in byte element i of VSR[VRB+32].
  - src1 is compared to src2. The larger of the two values is placed into byte element i of VSR[VRT+32].

**Special Registers Altered:**
None

**Register Data Layout for vxmaxsb & vxmaxub**

```
```

**Vector Maximum Unsigned Byte VX-form**

<table>
<thead>
<tr>
<th>vxmaxub</th>
<th>VRT, VRA, VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>VRT</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

- if MSR.VEC = 0 then Vector_Unavailable()
- do i = 0 to 15
  - src1 ← VSR[VRA+32].byte[i]
  - src2 ← VSR[VRB+32].byte[i]
  - gt_flag ← EXTZ(src1) > EXTZ(src2)
  - VSR[VRT+32].byte[i] ← gt_flag=1 ? src1 : src2
- end

For each integer value i from 0 to 15, do the following.
- Let src1 be the signed integer value in byte element i of VSR[VRA+32].
- Let src2 be the signed integer value in byte element i of VSR[VRB+32].
  - src1 is compared to src2. The larger of the two values is placed into byte element i of VSR[VRT+32].

**Special Registers Altered:**
None
Vector Maximum Signed Halfword VX-form

vmaxsh VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>32</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7

src1 ← VSR[VRA+32].hword[i]
src2 ← VSR[VRB+32].hword[i]

gt_flag ← EXTS(src1) > EXTS(src2)

VSR[VRT+32].hword[i] ← gt_flag=1 ? src1 : src2
end

For each integer value i from 0 to 7, do the following.

Let src1 be the signed integer value in halfword element i of VSR[VRA+32].

Let src2 be the signed integer value in halfword element i of VSR[VRB+32].

src1 is compared to src2. The larger of the two values is placed into halfword element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmaxsh & vmaxuh


0 1 16 32 48 64 80 96 112 127

Vector Maximum Unsigned Halfword VX-form

vmaxuh VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>66</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7

src1 ← VSR[VRA+32].hword[i]
src2 ← VSR[VRB+32].hword[i]

gt_flag ← EXTZ(src1) > EXTZ(src2)

VSR[VRT+32].hword[i] ← gt_flag=1 ? src1 : src2
end

For each integer value i from 0 to 7, do the following.

Let src1 be the unsigned integer value in halfword element i of VSR[VRA+32].

Let src2 be the unsigned integer value in halfword element i of VSR[VRB+32].

src1 is compared to src2. The larger of the two values is placed into halfword element i of VSR[VRT+32].

Special Registers Altered:

None
**Vector Maximum Signed Word VX-form**

vmaxsw

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>386</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3

    src1 ← VSR[VRA+32].word[i]
    src2 ← VSR[VRB+32].word[i]

    gt_flag ← EXTS(src1) > EXTS(src2)
    VSR[VRT+32].word[i] ← gt_flag=1 ? src1 : src2

end

For each integer value i from 0 to 3, do the following.
Let src1 be the signed integer value in word element i of VSR[VRA+32].
Let src2 be the signed integer value in word element i of VSR[VRB+32].
src1 is compared to src2. The larger of the two values is placed into word element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vmaxsw & vmaxuw

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
<td>127</td>
</tr>
</tbody>
</table>

**Vector Maximum Unsigned Word VX-form**

vmaxuw

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>130</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3

    src1 ← VSR[VRA+32].word[i]
    src2 ← VSR[VRB+32].word[i]

    gt_flag ← EXTZ(src1) > EXTZ(src2)
    VSR[VRT+32].word[i] ← gt_flag=1 ? src1 : src2

end

For each integer value i from 0 to 3, do the following.
Let src1 be the unsigned integer value in word element i of VSR[VRA+32].
Let src2 be the unsigned integer value in word element i of VSR[VRB+32].
src1 is compared to src2. The larger of the two values is placed into word element i of VSR[VRT+32].

Special Registers Altered:
None
Vector Maximum Signed Doubleword VX-form

vmaxsd VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 1
src1 ← VSR[VRA+32].dword[i]
src2 ← VSR[VRB+32].dword[i]
gt_flag ← EEXTS(src1) > EEXTS(src2)
VSR[VRT+32].dword[i] ← gt_flag=1 ? src1 : src2
end

For each integer value i from 0 to 1, do the following.
Let src1 be the signed integer value in doubleword element i of VSR[VRA+32].
Let src2 be the signed integer value in doubleword element i of VSR[VRB+32].
src1 is compared to src2. The larger of the two values is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vmaxsd & vmaxud

| src1 | VSR[VRA+32].dword[0] | VSR[VRA+32].dword[1] |
| src2 | VSR[VRB+32].dword[0] | VSR[VRB+32].dword[1] |
| result | VSR[VRT+32].dword[0] | VSR[VRT+32].dword[1] |

Vector Maximum Unsigned Doubleword VX-form

vmaxud VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 1
src1 ← VSR[VRA+32].dword[i]
src2 ← VSR[VRB+32].dword[i]
gt_flag ← EEXTZ(src1) > EEXTZ(src2)
VSR[VRT+32].dword[i] ← gt_flag=1 ? src1 : src2
end

For each integer value i from 0 to 1, do the following.
Let src1 be the unsigned integer value in doubleword element i of VSR[VRA+32].
Let src2 be the unsigned integer value in doubleword element i of VSR[VRB+32].
src1 is compared to src2. The larger of the two values is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:
None
### 6.9.2.2 Vector Integer Minimum Instructions

#### Vector Minimum Signed Byte VX-form

**vminsb**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>770</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

For each integer value \(i\) from 0 to 15, do the following:

- Let \(\text{src1}\) be the signed integer value in byte element \(i\) of VSR[VRA+32].
- Let \(\text{src2}\) be the signed integer value in byte element \(i\) of VSR[VRB+32].

\(\text{src1}\) is compared to \(\text{src2}\). The smaller of the two values is placed into byte element \(i\) of VSR[VRT+32].

**Special Registers Altered:** None

**Register Data Layout for vminsb & vminub**

|------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|

#### Vector Minimum Unsigned Byte VX-form

**vminub**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>514</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

For each integer value \(i\) from 0 to 15, do the following:

- Let \(\text{src1}\) be the unsigned integer value in byte element \(i\) of VSR[VRA+32].
- Let \(\text{src2}\) be the unsigned integer value in byte element \(i\) of VSR[VRB+32].

\(\text{src1}\) is compared to \(\text{src2}\). The smaller of the two values is placed into byte element \(i\) of VSR[VRT+32].

**Special Registers Altered:** None
Vector Minimum Signed Halfword VX-form

\[
vminsh \text{ VRT, VRA, VRB}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 \\
\hline
0 & 834 & 0 & 6 & 11 \\
\hline
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable();
do i = 0 to 7
\[
src1 \leftarrow VSR[VRA+32].hword[i]
\]
\[
src2 \leftarrow VSR[VRB+32].hword[i]
\]
\[
\text{lt_flag} \leftarrow \text{EXTS}(src1) < \text{EXTS}(src2)
\]
\[
VSR[VRT+32].hword[i] \leftarrow \text{lt_flag}=1 ? src1 : src2
\]
end

For each integer value \(i\) from 0 to 7, do the following.
Let \(\text{src1}\) be the signed integer value in halfword element \(i\) of \(VSR[VRA+32]\).
Let \(\text{src2}\) be the signed integer value in halfword element \(i\) of \(VSR[VRB+32]\).
\(\text{src1}\) is compared to \(\text{src2}\). The smaller of the two values is placed into halfword element \(i\) of \(VSR[VRT+32]\).

Special Registers Altered:
None

Register Data Layout for vminsh & vminuh

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
\hline
\hline
\hline
\end{array}
\]

Vector Minimum Unsigned Halfword VX-form

\[
vminuh \text{ VRT, VRA, VRB}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 \\
\hline
0 & 578 & 0 & 6 & 11 \\
\hline
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable();
do i = 0 to 7
\[
src1 \leftarrow VSR[VRA+32].hword[i]
\]
\[
src2 \leftarrow VSR[VRB+32].hword[i]
\]
\[
\text{lt_flag} \leftarrow \text{EXTZ}(src1) < \text{EXTZ}(src2)
\]
\[
VSR[VRT+32].hword[i] \leftarrow \text{lt_flag}=1 ? src1 : src2
\]
end

For each integer value \(i\) from 0 to 7, do the following.
Let \(\text{src1}\) be the unsigned integer value in halfword element \(i\) of \(VSR[VRA+32]\).
Let \(\text{src2}\) be the unsigned integer value in halfword element \(i\) of \(VSR[VRB+32]\).
\(\text{src1}\) is compared to \(\text{src2}\). The smaller of the two values is placed into halfword element \(i\) of \(VSR[VRT+32]\).

Special Registers Altered:
None
**Vector Minimum Signed Word VX-form**

```
vminsw VRT,VRA,VRB
```

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 3 \)
  - \( \text{src1 } \leftarrow \text{VSR}[VRA+32].\text{word}[i] \)
  - \( \text{src2 } \leftarrow \text{VSR}[VRB+32].\text{word}[i] \)
  - \( \text{lt_flag } \leftarrow \text{EXTS(src1) < EXTS(src2)} \)
  - \( \text{VSR}[VRT+32].\text{word}[i] \leftarrow \text{lt_flag=1 ? src1 : src2} \)
- \( \text{end} \)

For each integer value \( i \) from 0 to 3, do the following.

Let \( \text{src1} \) be the signed integer value in word element \( i \) of VSR[VRB+32].

Let \( \text{src2} \) be the signed integer value in word element \( i \) of VSR[VRB+32].

\( \text{src1} \) is compared to \( \text{src2} \). The smaller of the two values is placed into word element \( i \) of VSR[VRT+32].

**Special Registers Altered:**

None

**Register Data Layout for vminsw & vminuw**

|------|---------------------|---------------------|---------------------|---------------------|

**Vector Minimum Unsigned Word VX-form**

```
vminuw VRT,VRA,VRB
```

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 3 \)
  - \( \text{src1 } \leftarrow \text{VSR}[VRA+32].\text{word}[i] \)
  - \( \text{src2 } \leftarrow \text{VSR}[VRB+32].\text{word}[i] \)
  - \( \text{lt_flag } \leftarrow \text{EXTZ(src1) < EXTZ(src2)} \)
  - \( \text{VSR}[VRT+32].\text{word}[i] \leftarrow \text{lt_flag=1 ? src1 : src2} \)
- \( \text{end} \)

For each integer value \( i \) from 0 to 3, do the following.

Let \( \text{src1} \) be the unsigned integer value in word element \( i \) of VSR[VRB+32].

Let \( \text{src2} \) be the unsigned integer value in word element \( i \) of VSR[VRB+32].

\( \text{src1} \) is compared to \( \text{src2} \). The smaller of the two values is placed into word element \( i \) of VSR[VRT+32].

**Special Registers Altered:**

None
### Vector Minimum Signed Doubleword VX-form

**vminsd**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>962</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 1 \\
\text{src1} &\leftarrow \text{VSR}[\text{VRA+32}].\text{dword}[i] \\
\text{src2} &\leftarrow \text{VSR}[\text{VRB+32}].\text{dword}[i] \\
\text{lt_flag} &\leftarrow \text{EXTS}(\text{src1}) < \text{EXTS}(\text{src2}) \\
\text{VSR}[\text{VRT+32}].\text{dword}[i] &\leftarrow \text{lt_flag}=1 \ ? \ \text{src1} : \ \text{src2}
\end{align*}
\]

For each integer value \(i\) from 0 to 1, do the following.

Let \(\text{src1}\) be the signed integer value in doubleword element \(i\) of \(\text{VSR}[\text{VRA+32}]\).

Let \(\text{src2}\) be the signed integer value in doubleword element \(i\) of \(\text{VSR}[\text{VRB+32}]\).

\(\text{src1}\) is compared to \(\text{src2}\). The smaller of the two values is placed into doubleword element \(i\) of \(\text{VSR}[\text{VRT+32}]\).

**Special Registers Altered:**

None

### Vector Minimum Unsigned Doubleword VX-form

**vminud**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>706</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 1 \\
\text{src1} &\leftarrow \text{VSR}[\text{VRA+32}].\text{dword}[i] \\
\text{src2} &\leftarrow \text{VSR}[\text{VRB+32}].\text{dword}[i] \\
\text{lt_flag} &\leftarrow \text{EXTZ}(\text{src1}) < \text{EXTZ}(\text{src2}) \\
\text{VSR}[\text{VRT+32}].\text{dword}[i] &\leftarrow \text{lt_flag}=1 \ ? \ \text{src1} : \ \text{src2}
\end{align*}
\]

For each integer value \(i\) from 0 to 1, do the following.

Let \(\text{src1}\) be the unsigned integer value in doubleword element \(i\) of \(\text{VSR}[\text{VRA+32}]\).

Let \(\text{src2}\) be the unsigned integer value in doubleword element \(i\) of \(\text{VSR}[\text{VRB+32}]\).

\(\text{src1}\) is compared to \(\text{src2}\). The smaller of the two values is placed into doubleword element \(i\) of \(\text{VSR}[\text{VRT+32}]\).

**Special Registers Altered:**

None

### Register Data Layout for vminsd & vminud

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[\text{VRA+32}].\text{dword}[0]</td>
<td>VSR[\text{VRB+32}].\text{dword}[0]</td>
<td>VSR[\text{VRT+32}].\text{dword}[0]</td>
</tr>
<tr>
<td>VSR[\text{VRA+32}].\text{dword}[1]</td>
<td>VSR[\text{VRB+32}].\text{dword}[1]</td>
<td>VSR[\text{VRT+32}].\text{dword}[1]</td>
</tr>
</tbody>
</table>
6.9.3 Vector Integer Compare Instructions

The Vector Integer Compare instructions compare two VSRs element by element, interpreting the elements as unsigned or signed-integers depending on the instruction, and set the corresponding element of the target VSR to all ‘1s if the relation being tested is true and to all ‘0s if the relation being tested is false.

If Rc=1 CR Field 6 is set to reflect the result of the comparison, as follows.

---

**Programming Note**

vcmpqub[, vcmpquh[, vcmpquw[, and vcmpqud[], can be used for unsigned or signed-integers.

---

**Vector Compare Equal Unsigned Byte VC-form**

<table>
<thead>
<tr>
<th>Vcmpequb</th>
<th>VRT, VRA, VRB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vcmpequb</td>
<td>VRT, VRA, VRB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The relation is true for all element pairs (i.e., VSR[VRT+32] is set to all 1s)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>The relation is false for all element pairs (i.e., VSR[VRT+32] is set to all 0s)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

For each integer value \(i\) from 0 to 15, do the following.

Let \(src1\) be the unsigned integer value in byte element \(i\) of VSR[VRA+32].

Let \(src2\) be the unsigned integer value in byte element \(i\) of VSR[VRB+32].

\(src1\) is compared to \(src2\).

The contents of byte element \(i\) of VSR[VRT+32] are set to all 1s if \(src1\) is equal to \(src2\), and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

**Special Registers Altered:**

CR Field 6................................. (if Rc=1)

---

**Register Data Layout for vcmpequb[]**

|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127
Vector Compare Equal Unsigned Halfword VC-form

\begin{verbatim}
vcmpequh VRT,VRA,VRB (Rc=0) vcmpequh VRT,VRA,VRB (Rc=1)
\end{verbatim}

\begin{verbatim}
if MSR.VEC=0 then Vector_Unavailable()
all_true ← 1
all_false ← 1
do i = 0 to 7
  src1 ← VSR[VRA+32].hword[i]
  src2 ← VSR[VRB+32].hword[i]
  if src1 = src2 then do
    VSR[VRT+32].hword[i] ← 0xFFFF
    all_false ← 0
  end
  else do
    VSR[VRT+32].hword[i] ← 0x0000
    all_true ← 0
  end
end
if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)
\end{verbatim}

For each integer value $i$ from 0 to 7, do the following.

Let $src1$ be the unsigned integer value in halfword element $i$ of $VSR[VRA+32]$.

Let $src2$ be the unsigned integer value in halfword element $i$ of $VSR[VRB+32]$.

$src1$ is compared to $src2$.

The contents of halfword element $i$ of $VSR[VRT+32]$ are set to all 1s if $src1$ is equal to $src2$, and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

| CR field 6 | . . . . . . . . . . . . . . . . . . . . . . . . . . (if Rc=1) |

<table>
<thead>
<tr>
<th>Register Data Layout for vcmpequh[]</th>
</tr>
</thead>
</table>
Vector Compare Equal Unsigned Word VC-form

vcmpnequw  VRT,VRA,VRB  (Rc=0)
vcmpnequw  VRT,VRA,VRB  (Rc=1)

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>Rc</th>
<th>134</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable() end

all_true ← 1
all_false ← 1

do i = 0 to 3
src1 ← VSR[VRA+32].word[i]
src2 ← VSR[VRB+32].word[i]
if src1 = src2 then do
  VSR[VRT+32].word[i] ← 0xFFFF_FFFF
  all_false ← 0
end else do
  VSR[VRT+32].word[i] ← 0x0000_0000
  all_true ← 0
end
end

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

For each integer value i from 0 to 3, do the following.
Let src1 be the unsigned integer value in word element i of VSR[VRA+32].

Let src2 be the unsigned integer value in word element i of VSR[VRB+32].

c src1 is compared to src2.

The contents of word element i of VSR[VRT+32] are set to all 1s if src1 is equal to src2, and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 .............................. (if Rc=1)

Register Data Layout for vcmpnequw[]

|------|---------------------|---------------------|---------------------|---------------------|

0 32 64 96 127
Vector Compare Equal Unsigned Doubleword VC-form

vcompedu VRT, VRA, VRB (Rc=0)
vcomped VRT, VRA, VRB (Rc=1)

if MSR.VEC=0 then Vector_Unavailable()
all_true ← 1
all_false ← 1
do i = 0 to 1
src1 ← VSR[VRA+32].dword[i]
src2 ← VSR[VRB+32].dword[i]
if src1 = src2 then do
VSR[VRT+32].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
all_false ← 0
end
else do
VSR[VRT+32].dword[i] ← 0x0000_0000_0000_0000
all_true ← 0
end
end
if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

For each integer value i from 0 to 1, do the following.

Let src1 be the unsigned integer value in doubleword element i of VSR[VRA+32].

Let src2 be the unsigned integer value in doubleword element i of VSR[VRB+32].

src1 is compared to src2.

The contents of doubleword element i of VSR[VRT+32] are set to all 1s if src1 is equal to src2, and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 ......................... (if Rc=1)

Register Data Layout for vcompedu[]

<table>
<thead>
<tr>
<th></th>
<th>VSR[VRT+32].dword[0]</th>
<th>VSR[VRT+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td>VSR[VRA+32].dword[0]</td>
<td>VSR[VRA+32].dword[1]</td>
</tr>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
</tbody>
</table>
Vector Compare Equal Quadword VC-form

\[
\begin{align*}
\text{vcmpequq} & \quad \text{VRT},\text{VRA},\text{VRB} \quad \text{(Rc=0)} \\
\text{vcmpequq} & \quad \text{VRT},\text{VRA},\text{VRB} \quad \text{(Rc=1)}
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>Rc</th>
<th>455</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>22</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();

all_true ← 1
all_false ← 1
src1 ← VSR\[VRA+32\]
src2 ← VSR\[VRB+32\]

if src1 = src2 then do
  VSR\[VRT+32\] ← 0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
  all_false ← 0
end
else do
  VSR\[VRT+32\] ← 0x0000_0000_0000_0000_0000_0000_0000_0000
  all_true ← 0
end

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

Let \( src1 \) be the unsigned integer value in VSR\[VRA+32\].
Let \( src2 \) be the unsigned integer value in VSR\[VRB+32\].

If \( src1 \) is equal to \( src2 \), set VSR\[VRT+32\] to all 1s.
Otherwise, set VSR\[VRT+32\] to all 0s.

Special Registers Altered:
CR field 6 \( \text{if} \ Rc = 1 \)

Register Data Layout for vcmpequq[]

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

0 127
Vector Compare Greater Than Signed Byte
VC-form

\[
vcmpgtsb \ VRT,VRA,VRB \quad (Rc=0)
\]
\[
vcmpgtsb. \ VRT,VRA,VRB \quad (Rc=1)
\]

if MSR.VEC=0 then Vector_Unavailable()

all_true ← 1
all_false ← 1
do i = 0 to 15
src1 ← EXTS(VSR[VRA+32].byte[i])
src2 ← EXTS(VSR[VRB+32].byte[i])

if src1 > src2 then do
VSR[VRT+32].byte[i] ← 0xFF
all_false ← 0
end
else do
VSR[VRT+32].byte[i] ← 0x00
all_true ← 0
end
done

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

For each integer value \( i \) from 0 to 15, do the following.
Let \( src1 \) be the unsigned integer value in byte element \( i \) of VSR[VRA+32].
Let \( src2 \) be the unsigned integer value in byte element \( i \) of VSR[VRB+32].
\( src1 \) is compared to \( src2 \).
The contents of byte element \( i \) of VSR[VRT+32] are set to all 1s if \( src1 \) is equal to \( src2 \), and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 .......................... (if Rc=1)

Register Data Layout for \( vcmpgtsb[] \) & \( vcmpgtub[] \)

\[
\begin{array}{ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
Vector Compare Greater Than Signed Halfword VC-form

\[
\text{VCMPTGSH} \hspace{0.5cm} \text{VRT, VRA, VRB} \quad (Rc=0)
\]

\[
\text{VCMPTGSH.} \hspace{0.5cm} \text{VRT, VRA, VRB} \quad (Rc=1)
\]

<table>
<thead>
<tr>
<th>Rc</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>838</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>22</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

\begin{align*}
\text{if MSR.VEC=0 then Vector_Unavailable(\text{|}))}
\end{align*}

\begin{align*}
\text{all_true} & \leftarrow 1 \\
\text{all_false} & \leftarrow 1 \\
\text{do i = 0 to 7} \\
& \text{src1} \leftarrow \text{EXTS(VSR[VRA+32].hword[i])} \\
& \text{src2} \leftarrow \text{EXTS(VSR[VRB+32].hword[i])} \\
\text{if src1 > src2 then do} \\
& \text{VSR[VRT+32].hword[i]} \leftarrow 0xFFFF \\
& \text{all_false} \leftarrow 0 \\
\text{else do} \\
& \text{VSR[VRT+32].hword[i]} \leftarrow 0x0000 \\
& \text{all_true} \leftarrow 0 \\
\text{end} \\
\text{end}
\end{align*}

\begin{align*}
\text{if Rc=1 then \text{CR.bit[56:59]} \leftarrow \text{all_true << 3} + \text{all_false << 1}}
\end{align*}

For each integer value \(i\) from 0 to 7, do the following.

Let \(\text{src1}\) be the signed integer value in halfword element \(i\) of \(\text{VSR[VRA+32]}\).

Let \(\text{src2}\) be the signed integer value in halfword element \(i\) of \(\text{VSR[VRB+32]}\).

\(\text{src1}\) is compared to \(\text{src2}\).

The contents of halfword element \(i\) of \(\text{VSR[VRT+32]}\) are set to all 1s if \(\text{src1}\) is greater than \(\text{src2}\), and is set to all 0s otherwise.

If \(\text{Rc=1}\), CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

\begin{align*}
\text{CR field 6} \quad \text{............... (if Rc=1)}
\end{align*}

Special Registers Altered:

\begin{align*}
\text{CR field 6} \quad \text{............... (if Rc=1)}
\end{align*}

Register Data Layout for \(\text{vcmpgtsh[.]} \) & \(\text{vcmpgtuh[.]} \)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
<td>64</td>
<td>80</td>
<td>96</td>
<td>112</td>
<td>127</td>
</tr>
</tbody>
</table>

Vector Compare Greater Than Unsigned Halfword VC-form

\[
\text{VCMPTGUH} \hspace{0.5cm} \text{VRT, VRA, VRB} \quad (Rc=0)
\]

\[
\text{VCMPTGUH.} \hspace{0.5cm} \text{VRT, VRA, VRB} \quad (Rc=1)
\]

<table>
<thead>
<tr>
<th>Rc</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>582</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>22</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

\begin{align*}
\text{if MSR.VEC=0 then Vector_Unavailable(\text{|}))}
\end{align*}

\begin{align*}
\text{all_true} & \leftarrow 1 \\
\text{all_false} & \leftarrow 1 \\
\text{do i = 0 to 7} \\
& \text{src1} \leftarrow \text{EXTZ(VSR[VRA+32].hword[i])} \\
& \text{src2} \leftarrow \text{EXTZ(VSR[VRB+32].hword[i])} \\
\text{if src1 > src2 then do} \\
& \text{VSR[VRT+32].hword[i]} \leftarrow 0xFFFF \\
& \text{all_false} \leftarrow 0 \\
\text{else do} \\
& \text{VSR[VRT+32].hword[i]} \leftarrow 0x0000 \\
& \text{all_true} \leftarrow 0 \\
\text{end} \\
\text{end}
\end{align*}

\begin{align*}
\text{if Rc=1 then \text{CR.bit[56:59]} \leftarrow \text{all_true << 3} + \text{all_false << 1}}
\end{align*}

For each integer value \(i\) from 0 to 7, do the following.

Let \(\text{src1}\) be the unsigned integer value in halfword element \(i\) of \(\text{VSR[VRA+32]}\).

Let \(\text{src2}\) be the unsigned integer value in halfword element \(i\) of \(\text{VSR[VRB+32]}\).

\(\text{src1}\) is compared to \(\text{src2}\).

The contents of halfword element \(i\) of \(\text{VSR[VRT+32]}\) are set to all 1s if \(\text{src1}\) is greater than \(\text{src2}\), and is set to all 0s otherwise.

If \(\text{Rc=1}\), CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

\begin{align*}
\text{CR field 6} \quad \text{............... (if Rc=1)}
\end{align*}

Special Registers Altered:

\begin{align*}
\text{CR field 6} \quad \text{............... (if Rc=1)}
\end{align*}

Register Data Layout for \(\text{vcmpgtuh[.]} \)
Vector Compare Greater Than Signed Word
VC-form

**VC-form**

\[
\text{vcmpgtsw} \ VRT, VRA, VRB \ (Rc=0) \\
\text{vcmpgtsw.} \ VRT, VRA, VRB \ (Rc=1)
\]

if MSR.VEC=0 then Vector_Unavailable()

all_true ← 1
all_false ← 1

do i = 0 to 3
src1 ← EXTS(VSR[VRA+32].word[i])
src2 ← EXTS(VSR[VRB+32].word[i])

if src1 > src2 then do
VSR[VRT+32].word[i] ← 0xFFFF_FFFF
all_false ← 0
end
else do
VSR[VRT+32].word[i] ← 0x0000_0000
all_true ← 0
end
end

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

For each integer value \(i\) from 0 to 3, do the following.
Let \(src1\) be the signed integer value in word element \(i\) of VSR[VRA+32].
Let \(src2\) be the signed integer value in word element \(i\) of VSR[VRB+32].
\(src1\) is compared to \(src2\).
The contents of word element \(i\) of VSR[VRT+32] are set to all 1s if \(src1\) is greater than \(src2\), and is set to all 0s otherwise.

If \(Rc=1\), CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 ............................. (if \(Rc=1\))

Register Data Layout for vcmpgtsw[] & vcmpgtuw[

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

Vector Compare Greater Than Unsigned Word
VC-form

**VC-form**

\[
\text{vcmpgtuw} \ VRT, VRA, VRB \ (Rc=0) \\
\text{vcmpgtuw.} \ VRT, VRA, VRB \ (Rc=1)
\]

if MSR.VEC=0 then Vector_Unavailable()

all_true ← 1
all_false ← 1

do i = 0 to 3
src1 ← EXTZ(VSR[VRA+32].word[i])
src2 ← EXTZ(VSR[VRB+32].word[i])

if src1 > src2 then do
VSR[VRT+32].word[i] ← 0xFFFF_FFFF
all_false ← 0
end
else do
VSR[VRT+32].word[i] ← 0x0000_0000
all_true ← 0
end
end

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

For each integer value \(i\) from 0 to 3, do the following.
Let \(src1\) be the unsigned integer value in word element \(i\) of VSR[VRA+32].
Let \(src2\) be the unsigned integer value in word element \(i\) of VSR[VRB+32].
\(src1\) is compared to \(src2\).
The contents of word element \(i\) of VSR[VRT+32] are set to all 1s if \(src1\) is greater than \(src2\), and is set to all 0s otherwise.

If \(Rc=1\), CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 ............................. (if \(Rc=1\))

Register Data Layout for vcmpgtuw[]
Vector Compare Greater Than Signed Doubleword VC-form

\[ \text{vcmpgtsd } \text{VRT, VRA, VRB} \quad (\text{Rc} = 0) \]
\[ \text{vcmpgtsd. VRT, VRA, VRB} \quad (\text{Rc} = 1) \]

For each integer value \( i \) from 0 to 1, do the following.

Let \( \text{src1} \) be the signed integer value in doubleword element \( i \) of \( \text{VSR[VRA+32]} \).

Let \( \text{src2} \) be the signed integer value in doubleword element \( i \) of \( \text{VSR[VRB+32]} \).

\( \text{src1} \) is compared to \( \text{src2} \).

The contents of doubleword element \( i \) of \( \text{VSR[VRT+32]} \) are set to all 1s if \( \text{src1} \) is greater than \( \text{src2} \), and is set to all 0s otherwise.

If \( \text{Rc} = 1 \), CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

CR field 6 .............................................. (if \( \text{Rc} = 1 \))

Register Data Layout for vcmpgtsd[] & vcmpgtd[]

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
</tbody>
</table>

Vector Compare Greater Than Unsigned Doubleword VC-form

\[ \text{vcmpgtud } \text{VRT, VRA, VRB} \quad (\text{Rc} = 0) \]
\[ \text{vcmpgtud. VRT, VRA, VRB} \quad (\text{Rc} = 1) \]

For each integer value \( i \) from 0 to 1, do the following.

Let \( \text{src1} \) be the unsigned integer value in doubleword element \( i \) of \( \text{VSR[VRA+32]} \).

Let \( \text{src2} \) be the unsigned integer value in doubleword element \( i \) of \( \text{VSR[VRB+32]} \).

\( \text{src1} \) is compared to \( \text{src2} \).

The contents of doubleword element \( i \) of \( \text{VSR[VRT+32]} \) are set to all 1s if \( \text{src1} \) is greater than \( \text{src2} \), and is set to all 0s otherwise.

If \( \text{Rc} = 1 \), CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

CR field 6 .............................................. (if \( \text{Rc} = 1 \))
Vector Compare Greater Than Signed Quadword VC-form

\[
v\text{cmpgt}\quad \text{VRT, VRA, VRB} \quad (Rc=0)
\]
\[
v\text{cmpgt}\quad \text{VRT, VRA, VRB} \quad (Rc=1)
\]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>Rc</th>
<th>903</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>212</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

all_true ← 1
all_false ← 1

src1 ← EXT(S(VSR[VRA+32])
src2 ← EXT(S(VSR[VRB+32])

if src1 > src2 then do
  VSR[VRT+32] ← 0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
  all_false ← 0
end
else do
  VSR[VRT+32] ← 0x0000_0000_0000_0000_0000_0000_0000_0000
  all_true ← 0
end

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

Let src1 be the signed integer value in VSR[VRA+32].
Let src2 be the signed integer value in VSR[VRB+32].

src1 is compared to src2.

The contents of VSR[VRT+32] are set to all 1s if src1 is
greater than src2, and are set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate if src1 is
greater than src2 or if src1 is not greater than src2.

Special Registers Altered:

CR field 6 \hspace{1cm} (if Rc=1)

Register Data Layout for vcmpgtsq[] & vcmpgtuq[]

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
Vector Compare Not Equal Byte VC-form

\[ \text{vcmpneb} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (Rc=0) \]
\[ \text{vcmpneb.} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (Rc=1) \]

if MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{all}_{\text{true}} & \leftarrow 1 \\
\text{all}_{\text{false}} & \leftarrow 1 \\
\text{do} \ i = 0 \ \text{to} \ 15 \\
\text{src1} & \leftarrow \text{VSR[VRT+32].byte[i]} \\
\text{src2} & \leftarrow \text{VSR[VRB+32].byte[i]} \\
\text{if} \ \text{src1} \neq \text{src2} \ \text{then do} \\
& \quad \text{VSR[VRT+32].byte[i]} \leftarrow 0xFF \\
& \quad \text{all}_{\text{false}} \leftarrow 0 \\
\text{else do} \\
& \quad \text{VSR[VRT+32].byte[i]} \leftarrow 0x00 \\
& \quad \text{all}_{\text{true}} \leftarrow 0 \\
\text{end} \\
\end{align*}
\]

if Rc=1 then CR.bit[56:59] \leftarrow (\text{all}_{\text{true}}\times3) + (\text{all}_{\text{false}}\times1)

For each integer value \( i \) from 0 to 15, do the following.

Let \( \text{src1} \) be the contents of byte element \( i \) of \( \text{VSR[VRT+32]} \).

Let \( \text{src2} \) be the contents of byte element \( i \) of \( \text{VSR[VRB+32]} \).

\( \text{src1} \) is compared to \( \text{src2} \).

The contents of byte element \( i \) of \( \text{VSR[VRT+32]} \) are set to all 1s if \( \text{src1} \) is not equal to \( \text{src2} \), and are set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 (if Rc=1)

Register Data Layout for vcmpneb{} \& vcmpnezb{}:

<table>
<thead>
<tr>
<th>src1</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[0]</td>
<td>byte[15]</td>
</tr>
<tr>
<td>byte[1]</td>
<td>byte[14]</td>
</tr>
<tr>
<td>byte[15]</td>
<td>byte[0]</td>
</tr>
</tbody>
</table>

Vector Compare Not Equal or Zero Byte VC-form

\[ \text{vcmpnezb} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (Rc=0) \]
\[ \text{vcmpnezb.} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (Rc=1) \]

if MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{all}_{\text{true}} & \leftarrow 1 \\
\text{all}_{\text{false}} & \leftarrow 1 \\
\text{do} \ i = 0 \ \text{to} \ 15 \\
\text{src1} & \leftarrow \text{VSR[VRT+32].byte[i]} \\
\text{src2} & \leftarrow \text{VSR[VRB+32].byte[i]} \\
\text{if} \ \text{src1} = 0 \ | \ \text{src2} = 0 \ | \ \text{src1} \neq \text{src2} \ \text{then do} \\
& \quad \text{VSR[VRT+32].byte[i]} \leftarrow 0xFF \\
& \quad \text{all}_{\text{false}} \leftarrow 0 \\
\text{else do} \\
& \quad \text{VSR[VRT+32].byte[i]} \leftarrow 0x00 \\
& \quad \text{all}_{\text{true}} \leftarrow 0 \\
\text{end} \\
\end{align*}
\]

if Rc=1 then CR.bit[56:59] \leftarrow (\text{all}_{\text{true}}\times2) + (\text{all}_{\text{false}}\times1)

For each integer value \( i \) from 0 to 15, do the following.

Let \( \text{src1} \) be the contents of byte element \( i \) of \( \text{VSR[VRT+32]} \).

Let \( \text{src2} \) be the contents of byte element \( i \) of \( \text{VSR[VRB+32]} \).

\( \text{src1} \) is compared to \( \text{src2} \).

The contents of byte element \( i \) of \( \text{VSR[VRT+32]} \) are set to all 1s if \( \text{src1} \) is not equal to \( \text{src2} \) or either \( \text{src1} \) or \( \text{src2} \) is equal to 0x00, and are set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 (if Rc=1)
Vector Compare Not Equal Halfword VC-form

\[ \text{vcmpneh} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (Rc=0) \]
\[ \text{vcmpneh.} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (Rc=1) \]

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>Rc</th>
<th>71</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

all_true ← 1
all_false ← 1

do i = 0 to 7
  src1 ← VSR[VRA+32].hword[i]
  src2 ← VSR[VRB+32].hword[i]
  if src1 != src2 then do
    VSR[VRT+32].hword[i] ← 0xFFFF
    all_false ← 0
  end
  else do
    VSR[VRT+32].hword[i] ← 0x0000
    all_true ← 0
  end
endo

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

For each integer value \( i \) from 0 to 7, do the following.

Let \( src1 \) be the contents of halfword element \( i \) of \( \text{VSR}[\text{VRA}+32] \).

Let \( src2 \) be the contents of halfword element \( i \) of \( \text{VSR}[\text{VRB}+32] \).

\( src1 \) is compared to \( src2 \).

The contents of halfword element \( i \) of \( \text{VSR}[\text{VRT}+32] \) are set to all 1s if \( src1 \) is not equal to \( src2 \), and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6

Special Registers Altered:
CR field 6

Register Data Layout for vcmpneh[.] & vcmpnezh[.]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
<td>64</td>
<td>80</td>
<td>96</td>
<td>112</td>
<td>127</td>
</tr>
</tbody>
</table>
Vector Compare Not Equal Word VC-form

\[ \text{vcmpnew} \ VRT,\text{VRA},\text{VRB} \quad (Rc=0) \]
\[ \text{vcmpnew} \ . \ VRT,\text{VRA},\text{VRB} \quad (Rc=1) \]

\[
\begin{array}{cccccc}
4 & VRT & VRA & VRB & Rc & 135 \\
0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable()

all_true ← 1
all_false ← 1

do \ i = 0 \ to \ 3

\[
\begin{array}{c}
\text{src1} \leftarrow \text{VSR}[\text{VRA}+32].\text{word[}\ i\ ] \\
\text{src2} \leftarrow \text{VSR}[\text{VRB}+32].\text{word[}\ i\ ] \\
\end{array}
\]

if src1 != src2 then do

\[
\begin{array}{c}
\text{VSR}[\text{VRT}+32].\text{word[}\ i\ ] \leftarrow \text{0xFFFF_FFFF} \\
\text{all_false} \leftarrow 0 \\
\end{array}
\]

else do

\[
\begin{array}{c}
\text{VSR}[\text{VRT}+32].\text{word[}\ i\ ] \leftarrow \text{0x0000_0000} \\
\text{all_true} \leftarrow 0 \\
\end{array}
\]

end
end

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

For each integer value \( i \) from 0 to 3, do the following.

Let \( \text{src1} \) be the contents of word element \( i \) of \( \text{VSR}[\text{VRA}+32] \).

Let \( \text{src2} \) be the contents of word element \( i \) of \( \text{VSR}[\text{VRB}+32] \).

\( \text{src1} \) is compared to \( \text{src2} \).

The contents of word element \( i \) of \( \text{VSR}[\text{VRT}+32] \) are set to all 1s if \( \text{src1} \) is not equal to \( \text{src2} \), and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 
(\( \text{if Rc=1} \))

Register Data Layout for vcmpnew[.] & vcmpnezw[.]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
<td>128</td>
</tr>
</tbody>
</table>
Vector Compare Signed Quadword VX-form

\texttt{vcmpsq BF,VRA,VRB}

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>9</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>321</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>BF</td>
<td>VRA</td>
<td>VRB</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\begin{itemize}
\item If MSR.VEC=0 then Vector_Unavailable()
\item src1 ← ETS(VSR[VRA+32])
\item src2 ← ETS(VSR[VRB+32])
\item lt_flag ← src1 < src2
\item gt_flag ← src1 > src2
\item eq_flag ← src1 = src2
\item CR.field[BF] ← lt_flag<<3 | gt_flag<<2 | eq_flag<<1
\end{itemize}

Let src1 be the signed integer value in VSR[VRA+32]. Let src2 be the signed integer value in VSR[VRB+32].

Compare src1 with src2, place the comparison flags into CR field BF.

Special Registers Altered:
CR field BF

Register Data Layout for vcmpsq[.] & vcmpuq[.]

\begin{itemize}
\item src1
\item src2
\end{itemize}

Vector Compare Unsigned Quadword VX-form

\texttt{vcmpuq BF,VRA,VRB}

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>9</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>257</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>BF</td>
<td>VRA</td>
<td>VRB</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\begin{itemize}
\item If MSR.VEC=0 then Vector_Unavailable()
\item src1 ← ETS(VSR[VRA+32])
\item src2 ← ETS(VSR[VRB+32])
\item lt_flag ← src1 < src2
\item gt_flag ← src1 > src2
\item eq_flag ← src1 = src2
\item CR.field[BF] ← lt_flag<<3 | gt_flag<<2 | eq_flag<<1
\end{itemize}

Let src1 be the unsigned integer value in VSR[VRA+32]. Let src2 be the unsigned integer value in VSR[VRB+32].

Compare src1 with src2, place the comparison flags into CR field BF.

Special Registers Altered:
CR field BF
6.9.4 Vector Logical Instructions

Extended mnemonics for vector logical operations

Extended mnemonics are provided that use the Vector OR and Vector NOR instructions to copy the contents of one VSR to another, with and without complementing. These are shown as examples with the two instructions.

Vector Move Register

Several vector instructions can be coded in a way such that they simply copy the contents of one VSR to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register Vy to register Vx.

\[ \text{vmr} \ Vx, Vy \quad \text{(equivalent to:} \quad \text{vor} \ Vx, Vy, Vy) \]

Vector Complement Register

The Vector NOR instruction can be coded in a way such that it complements the contents of one VSR and places the result into another VSR. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register Vy and places the result into register Vx.

\[ \text{vnot} \ Vx, Vy \quad \text{(equivalent to:} \quad \text{vnor} \ Vx, Vy, Vy) \]

Vector Logical AND VX-form

\[ \text{vand} \ VRT, VRA, VRB \]

\[
\begin{array}{cccc}
0 & 6 & 11 & 16 \\
4 & 192 & 21 & 31
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable();

\[ \text{VSR}[VRT+32] \leftarrow \text{VSR}[VRA+32] \land \text{VSR}[VRB+32] \]

The contents of VSR[VRA+32] are ANDed with the contents of VSR[VRB+32] and the result is placed into VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vand & vandc

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

Vector Logical AND with Complement VX-form

\[ \text{vandc} \ VRT, VRA, VRB \]

\[
\begin{array}{cccc}
0 & 6 & 11 & 16 \\
4 & 192 & 21 & 31
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable();

\[ \text{VSR}[VRT+32] \leftarrow \text{VSR}[VRA+32] \land \neg \text{VSR}[VRB+32] \]

The contents of VSR[VRA+32] are ANDed with the complement of the contents of VSR[VRB+32] and the result is placed into VSR[VRT+32].

Special Registers Altered:
None
### Vector Logical Equivalence VX-form

**veqv**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\text{VSR}[\text{VRT}+32] \leftarrow \text{VSR}[\text{VRA}+32] \oplus \text{VSR}[\text{VRB}+32]
\]

The contents of VSR[VRT+32] are XORed with the contents of VSR[VRB+32] and the complemented result is placed into VSR[VRT+32].

**Special Registers Altered:**

None

### Vector Logical NAND VX-form

**vnand**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\text{VSR}[\text{VRT}+32] \leftarrow \neg (\text{VSR}[\text{VRA}+32] \& \text{VSR}[\text{VRB}+32])
\]

The contents of VSR[VRA+32] are ANDed with the contents of VSR[VRB+32] and the complemented result is placed into VSR[VRT+32].

**Special Registers Altered:**

None

### Vector Logical OR VX-form

**vor**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\text{VSR}[\text{VRT}+32] \leftarrow \text{VSR}[\text{VRA}+32] \mid \text{VSR}[\text{VRB}+32]
\]

The contents of VSR[VRA+32] are ORed with the contents of VSR[VRB+32] and the result is placed into VSR[VRT+32].

**Special Registers Altered:**

None

### Vector Logical OR with Complement VX-form

**vorc**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\text{VSR}[\text{VRT}+32] \leftarrow \text{VSR}[\text{VRA}+32] \mid \neg \text{VSR}[\text{VRB}+32]
\]

The contents of VSR[VRA+32] are ORed with the complement of the contents of VSR[VRB+32] and the result is placed into VSR[VRT+32].

**Special Registers Altered:**

None

### Vector Logical NOR VX-form

**vnor**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\text{VSR}[\text{VRT}+32] \leftarrow \neg (\text{VSR}[\text{VRA}+32] \mid \text{VSR}[\text{VRB}+32])
\]

The contents of VSR[VRA+32] are ORed with the contents of VSR[VRB+32] and the complemented result is placed into VSR[VRT+32].

**Special Registers Altered:**

None

### Vector Logical XOR VX-form

**vxor**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\text{VSR}[\text{VRT}+32] \leftarrow \text{VSR}[\text{VRA}+32] \oplus \text{VSR}[\text{VRB}+32]
\]

The contents of VSR[VRA+32] are XORed with the contents of VSR[VRB+32] and the result is placed into VSR[VRT+32].

**Special Registers Altered:**

None

### Register Data Layout for veqv, vnand, vor, vorc, vnor & vxor

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
6.9.5 Vector Integer Rotate Instructions

6.9.5.1 Vector Integer Rotate Left Instructions

Vector Rotate Left Byte VX-form

```
vlrb VRT,VRA,VRB
```

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
    src ← VSR[VRA+32].byte[i]
    sh ← VSR[VRB+32].byte[i].bit[5:7]
    VSR[VRT+32].byte[i] ← src <<< sh
end
```

For each integer value i from 0 to 15, do the following.
Let src1 be the contents of byte element i of VSR[VRA+32].
Let src2 be the contents of byte element i of VSR[VRB+32].
src1 is rotated left by the number of bits specified in the low-order 3 bits of src2.
The result is placed into byte element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vlrb

```
<table>
<thead>
<tr>
<th>src1</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[0]</td>
<td>byte[0]</td>
</tr>
<tr>
<td>byte[8]</td>
<td>byte[8]</td>
</tr>
<tr>
<td>byte[12]</td>
<td>byte[12]</td>
</tr>
<tr>
<td>byte[14]</td>
<td>byte[14]</td>
</tr>
</tbody>
</table>
```

Register Data Layout for vlrb

```
<table>
<thead>
<tr>
<th>src1</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[0]</td>
<td>byte[0]</td>
</tr>
<tr>
<td>byte[8]</td>
<td>byte[8]</td>
</tr>
<tr>
<td>byte[12]</td>
<td>byte[12]</td>
</tr>
<tr>
<td>byte[14]</td>
<td>byte[14]</td>
</tr>
</tbody>
</table>
```

Vector Rotate Left Halfword VX-form

```
vrhs VRT,VRA,VRB
```

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
    src ← VSR[VRA+32].hword[i]
    sh ← VSR[VRB+32].hword[i].bit[12:15]
    VSR[VRT+32].hword[i] ← src <<< sh
end
```

For each integer value i from 0 to 7, do the following.
Let src1 be the contents of halfword element i of VSR[VRA+32].
Let src2 be the contents of halfword element i of VSR[VRB+32].
src1 is rotated left by the number of bits specified in the low-order 4 bits of src2.
The result is placed into halfword element i of VSR[VRT+32].

Special Registers Altered:
None
Vector Rotate Left Word VX-form

vrlw VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable();
do i = 0 to 3
   src ← VSR[VRA+32].word[i]
   sh ← VSR[VRB+32].word[i].bit[27:31]
   VSR[VRT+32].word[i] ← src <<< sh
end

For each integer value i from 0 to 3, do the following.
Let src1 be the contents of word element i of VSR[VRA+32].
Let src2 be the contents of word element i of VSR[VRB+32].
src1 is rotated left by the number of bits specified in the low-order 5 bits of src2.
The result is placed into word element i in VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vrlw

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 32 64 96 127</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector Rotate Left Doubleword VX-form

vrld VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable();
do i = 0 to 1
   src ← VSR[VRA+32].dword[i]
   sh ← VSR[VRB+32].dword[i].bit[58:63]
   VSR[VRT+32].dword[i] ← src <<< sh
end

For each integer value i from 0 to 1, do the following.
Let src1 be the contents of doubleword element i of VSR[VRA+32].
Let src2 be the contents of doubleword element i of VSR[VRB+32].
src1 is rotated left by the number of bits specified in the low-order 6 bits of src2.
The result is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vrld

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
<tr>
<td></td>
<td>0 64 127</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 6. Vector Facility  405
**Vector Rotate Left Quadword VX-form**

vrlq  VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>11</th>
<th>VRA</th>
<th>16</th>
<th>VRB</th>
<th>21</th>
<th>5</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[
n \leftarrow VSR[VRB+32].\text{bit}[57:63] \\
VSR[VRT+32] \leftarrow \text{ROTL128}(VSR[VRA+32], n)
\]

Let \( SH \) be the contents of bits 57:63 of \( VSR[VRB+32] \).

Let \( src1 \) be the contents of \( VSR[VRA+32] \).

\( src1 \) is rotated left by \( SH \) bits. Bits shifted out on the left are shifted in on the right to replace vacated bits.

Special Registers Altered:
None

**Register Data Layout for vrlq**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
6.9.5.2 Vector Integer Rotate Left then AND with Mask Instructions

### Vector Rotate Left Word then AND with Mask VX-form

`vrlwnm VRT,VRA,VRB`

For each integer value \( i \) from 0 to 3, do the following.

Let \( \text{src1} \) be the contents of word element \( i \) of \( \text{VSR}[\text{VRA}+32] \).

Let \( \text{src2} \) be the contents of word element \( i \) of \( \text{VSR}[\text{VRB}+32] \).

Let \( \text{mb} \) be the contents of bits 11:15 of \( \text{src2} \).

Let \( \text{me} \) be the contents of bits 19:23 of \( \text{src2} \).

Let \( \text{sh} \) be the contents of bits 27:31 of \( \text{src2} \).

\( \text{src1} \) is rotated left \( \text{sh} \) bits.

A mask is generated having 1-bits from bit \( \text{mb} \) through bit \( \text{me} \) and 0-bits elsewhere.

The rotated data are ANDed with the generated mask.

The result is placed into word element \( i \) of \( \text{VSR}[\text{VRT}+32] \).

**Special Registers Altered:**

None

### Register Data Layout for vrlwnm

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{VSR}[\text{VRA}+32].\text{word}[0] )</td>
<td>( \text{VSR}[\text{VRB}+32].\text{word}[0] )</td>
<td>( \text{VSR}[\text{VRT}+32].\text{word}[0] )</td>
</tr>
<tr>
<td>( \text{VSR}[\text{VRA}+32].\text{word}[1] )</td>
<td>( \text{VSR}[\text{VRB}+32].\text{word}[1] )</td>
<td>( \text{VSR}[\text{VRT}+32].\text{word}[1] )</td>
</tr>
<tr>
<td>( \text{VSR}[\text{VRA}+32].\text{word}[2] )</td>
<td>( \text{VSR}[\text{VRB}+32].\text{word}[2] )</td>
<td>( \text{VSR}[\text{VRT}+32].\text{word}[2] )</td>
</tr>
<tr>
<td>( \text{VSR}[\text{VRA}+32].\text{word}[3] )</td>
<td>( \text{VSR}[\text{VRB}+32].\text{word}[3] )</td>
<td>( \text{VSR}[\text{VRT}+32].\text{word}[3] )</td>
</tr>
</tbody>
</table>
Vector Rotate Left Doubleword then AND with Mask VX-form

If MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 1 \\
\text{src1}.\text{dword}[0] &\leftarrow \text{VSR}[\text{VRA+32}].\text{dword}[i] \\
\text{src1}.\text{dword}[1] &\leftarrow \text{VSR}[\text{VRA+32}].\text{dword}[i] \\
\text{src2} &\leftarrow \text{VSR}[\text{VRB+32}].\text{dword}[i] \\
\text{b} &\leftarrow \text{src2}.\text{bit}[42:47] \\
\text{e} &\leftarrow \text{src2}.\text{bit}[50:55] \\
\text{m} &\leftarrow \text{src2}.\text{bit}[58:63] \\
\text{r} &\leftarrow \text{src1}.\text{bit}[n:n+63] \\
\text{m} &\leftarrow \text{MASK}(\text{b, e}) \\
\text{VSR}[\text{VRT+32}].\text{dword}[i] &\leftarrow \text{r & m}
\end{align*}
\]

For each integer value \(i\) from 0 to 1, do the following.

Let \(\text{src1}\) be the contents of doubleword element \(i\) of \(\text{VSR}[\text{VRA+32}]\).

Let \(\text{src2}\) be the contents of doubleword element \(i\) of \(\text{VSR}[\text{VRB+32}]\).

Let \(\text{mb}\) be the contents of bits 42:47 of \(\text{src2}\).
Let \(\text{me}\) be the contents of bits 50:55 of \(\text{src2}\).
Let \(\text{sh}\) be the contents of bits 58:63 of \(\text{src2}\).

\(\text{src1}\) is rotated left \(\text{sh}\) bits.

A mask is generated having 1-bits from bit \(\text{mb}\) through bit \(\text{me}\) and 0-bits elsewhere.

The rotated data are ANDed with the generated mask.

The result is placed into doubleword element \(i\) of \(\text{VSR}[\text{VRT+32}]\).

Special Registers Altered:
None

Register Data Layout for \text{vrldnm}

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRB+32].dword[0]</th>
<th>VSR[VRB+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
</tbody>
</table>

0 64 127
**Vector Rotate Left Quadword then AND with Mask VX-form**

Let src1 be the contents of VSR[VRA+32].
Let src2 be the contents of VSR[VRB+32].
Let mb be the contents of bits 41:47 of src2.
Let me be the contents of bits 49:55 of src2.
Let sh be the contents of bits 57:63 of src2.

src1 is rotated left sh bits.

A mask is generated having 1-bits from bit mb through bit me and 0-bits elsewhere.

The rotated data are ANDed with the generated mask.

The result is placed into VSR[VRT+32].

**Special Registers Altered:**
None
6.9.5.3 Vector Integer Rotate Left then Mask Insert Instructions

**Vector Rotate Left Word then Mask Insert VX-form**

For each integer value $i$ from 0 to 3, do the following.

Let $src1$ be the contents of word element $i$ of $VSR[VRA+32]$.

Let $src2$ be the contents of word element $i$ of $VSR[VRB+32]$.

Let $src3$ be the contents of word element $i$ of $VSR[VRT+32]$.

Let $mb$ be the contents of bits 11:15 of $src2$.

Let $me$ be the contents of bits 19:23 of $src2$.

Let $sh$ be the contents of bits 27:31 of $src2$.

$src1$ is rotated left $sh$ bits.

A mask is generated having 1-bits from bit $mb$ through bit $me$ and 0-bits elsewhere.

The rotated data are inserted into $src3$ under control of the generated mask.

The result is placed into word element $i$ of $VSR[VRT+32]$.

**Special Registers Altered:**

None

---

**Register Data Layout for \text{vrlwmi}**

<table>
<thead>
<tr>
<th>vrlwmi</th>
<th>VRT, VRA, VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>VRT</td>
</tr>
<tr>
<td>2</td>
<td>VRA</td>
</tr>
<tr>
<td>3</td>
<td>VRB</td>
</tr>
<tr>
<td>4</td>
<td>133</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then VectorUnavailable()

do $i = 0$ to 3

$src1.\text{word}[0] \leftarrow VSR[VRA+32].\text{word}[i]$

$src1.\text{word}[1] \leftarrow VSR[VRA+32].\text{word}[i]$

$src2 \leftarrow VSR[VRB+32].\text{word}[i]$

$src3 \leftarrow VSR[VRT+32].\text{word}[i]$

$b \leftarrow src2.\text{bit}[11:15]$

e \leftarrow src2.\text{bit}[19:23]$

$h \leftarrow src2.\text{bit}[27:31]$

$r \leftarrow src1.\text{bit}[n:n+31]$

$mb$ is the contents of bits 11:15 of $src2$.

$me$ is the contents of bits 19:23 of $src2$.

$sh$ is the contents of bits 27:31 of $src2$.

$src1$ is rotated left $sh$ bits.

A mask is generated having 1-bits from bit $mb$ through bit $me$ and 0-bits elsewhere.

The rotated data are inserted into $src3$ under control of the generated mask.

The result is placed into word element $i$ of $VSR[VRT+32]$.

**Special Registers Altered:**

None

---

**Register Data Layout for vrlwmi**

<table>
<thead>
<tr>
<th>src1</th>
<th>$VSR[\text{VRA+32}].\text{word}[0]$</th>
<th>$VSR[\text{VRA+32}].\text{word}[1]$</th>
<th>$VSR[\text{VRA+32}].\text{word}[2]$</th>
<th>$VSR[\text{VRA+32}].\text{word}[3]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>$VSR[\text{VRB+32}].\text{word}[0]$</td>
<td>$VSR[\text{VRB+32}].\text{word}[1]$</td>
<td>$VSR[\text{VRB+32}].\text{word}[2]$</td>
<td>$VSR[\text{VRB+32}].\text{word}[3]$</td>
</tr>
<tr>
<td>result</td>
<td>$VSR[\text{VRT+32}].\text{word}[0]$</td>
<td>$VSR[\text{VRT+32}].\text{word}[1]$</td>
<td>$VSR[\text{VRT+32}].\text{word}[2]$</td>
<td>$VSR[\text{VRT+32}].\text{word}[3]$</td>
</tr>
</tbody>
</table>
Vector Rotate Left Doubleword then Mask Insert VX-form

vrldmi VRT,VRA,VRB

For each integer value \( i \) from 0 to 1, do the following:

Let \( \text{src1} \) be the contents of doubleword element \( i \) of \( \text{VSR}[\text{VRA+32}] \).

Let \( \text{src2} \) be the contents of doubleword element \( i \) of \( \text{VSR}[\text{VRB+32}] \).

Let \( \text{src3} \) be the contents of doubleword element \( i \) of \( \text{VSR}[\text{VRT+32}] \).

Let \( \text{mb} \) be the contents of bits 42:47 of \( \text{src2} \).

Let \( \text{me} \) be the contents of bits 50:55 of \( \text{src2} \).

Let \( \text{sh} \) be the contents of bits 58:63 of \( \text{src2} \).

\( \text{src1} \) is rotated left \( \text{sh} \) bits.

A mask is generated having 1-bits from bit \( \text{mb} \) through bit \( \text{me} \) and 0-bits elsewhere.

The rotated data are inserted into \( \text{src3} \) under control of the generated mask.

The result is placed into doubleword element \( i \) of \( \text{VSR}[\text{VRT+32}] \).

Special Registers Altered:
None

Register Data Layout for vrldmi

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \text{src1} )</th>
<th>( \text{src2} )</th>
<th>( \text{result} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \text{VSR}[\text{VRA+32}].\text{dword}[0] )</td>
<td>( \text{VSR}[\text{VRB+32}].\text{dword}[0] )</td>
<td>( \text{VSR}[\text{VRT+32}].\text{dword}[0] )</td>
</tr>
<tr>
<td>1</td>
<td>( \text{VSR}[\text{VRA+32}].\text{dword}[1] )</td>
<td>( \text{VSR}[\text{VRB+32}].\text{dword}[1] )</td>
<td>( \text{VSR}[\text{VRT+32}].\text{dword}[1] )</td>
</tr>
</tbody>
</table>
**Vector Rotate Left Quadword then Mask Insert VX-form**

\[ \text{vrlqmi} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>69</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

- if MSR.VEC=0 then Vector.Unavailable()
- \( b \leftarrow \text{VSR}[\text{VRB}+32].\text{bit}[41:47] \)
- \( e \leftarrow \text{VSR}[\text{VRB}+32].\text{bit}[49:55] \)
- \( n \leftarrow \text{VSR}[\text{VRB}+32].\text{bit}[57:63] \)
- \( r \leftarrow \text{ROTL128}(\text{VSR}[\text{VRA}+32], n) \)
- \( m \leftarrow \text{MASK128}(b, e) \)
- \( \text{VSR}[\text{VRT}+32] \leftarrow (r \& m) \mid (\text{VSR}[\text{VRT}+32] \& \sim m) \)

Let \( \text{src1} \) be the contents of \( \text{VSR}[\text{VRA}+32] \).
Let \( \text{src2} \) be the contents of \( \text{VSR}[\text{VRB}+32] \).
Let \( \text{src3} \) be the contents of \( \text{VSR}[\text{VRT}+32] \).

Let \( \text{mb} \) be the contents of bits 41:47 of \( \text{src2} \).
Let \( \text{me} \) be the contents of bits 49:55 of \( \text{src2} \).
Let \( \text{sh} \) be the contents of bits 57:63 of \( \text{src2} \).

\( \text{src1} \) is rotated left \( \text{sh} \) bits.

A mask is generated having 1-bits from bit \( \text{mb} \) through bit \( \text{me} \) and 0-bits elsewhere.

The rotated data are inserted into \( \text{src3} \) under control of the generated mask.

The result is placed into \( \text{VSR}[\text{VRT}+32] \).

**Special Registers Altered:**
- None

**Register Data Layout for vrlqmi**

- \( \text{src1} \)  \( \text{VSR}[\text{VRA}+32] \)
- \( \text{src2} \)  \( \text{VSR}[\text{VRB}+32] \)
- \( \text{result} \)  \( \text{VSR}[\text{VRT}+32] \)
6.9.6 Vector Integer Shift Instructions

6.9.6.1 Vector Integer Shift Left Instructions

Vector Shift Left Byte VX-form

\[ \text{vslb } \text{VRT, VRA, VRB} \]

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 15 \)
  - \( \text{src1} \leftarrow \text{VSR}[\text{VRA+32}].\text{byte}[i] \)
  - \( \text{src2} \leftarrow \text{VSR}[\text{VRB+32}].\text{byte}[i].\text{bit}[5:7] \)
  - \( \text{VSR}[\text{VRT+32}].\text{byte}[i] \leftarrow \text{src1} \ll \text{src2} \)
- \( \text{end} \)

For each integer value \( i \) from 0 to 15, do the following.

Let \( \text{src1} \) be the contents of byte element \( i \) of \( \text{VSR}[\text{VRA+32}] \).

Let \( \text{src2} \) be the contents of byte element \( i \) of \( \text{VSR}[\text{VRB+32}] \).

\( \text{src1} \) is shifted left by the number of bits specified in the low-order 3 bits of \( \text{src2} \).

- Bits shifted out the most-significant bit are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into byte element \( i \) of \( \text{VSR}[\text{VRT+32}] \).

Special Registers Altered:
None

Register Data Layout for \text{vslb}

\[
\begin{array}{cccccccccccccccc}
\text{src1} & & & & & & & & & & & & & & & & \\
\text{src2} & & & & & & & & & & & & & & & & \\
\text{result} & & & & & & & & & & & & & & & & \\
\end{array}
\]

Vector Shift Left Halfword VX-form

\[ \text{vslh } \text{VRT, VRA, VRB} \]

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 7 \)
  - \( \text{src1} \leftarrow \text{VSR}[\text{VRA+32}].\text{hword}[i] \)
  - \( \text{src2} \leftarrow \text{VSR}[\text{VRB+32}].\text{hword}[i].\text{bit}[12:15] \)
  - \( \text{VSR}[\text{VRT+32}].\text{hword}[i] \leftarrow \text{src1} \ll \text{src2} \)
- \( \text{end} \)

For each integer value \( i \) from 0 to 7, do the following.

Let \( \text{src1} \) be the contents of halfword element \( i \) of \( \text{VSR}[\text{VRA+32}] \).

Let \( \text{src2} \) be the contents of halfword element \( i \) of \( \text{VSR}[\text{VRB+32}] \).

\( \text{src1} \) is shifted left by the number of bits specified in the low-order 4 bits of \( \text{src2} \).

- Bits shifted out the most-significant bit are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into halfword element \( i \) of \( \text{VSR}[\text{VRT+32}] \).

Special Registers Altered:
None

Register Data Layout for \text{vslh}

\[
\begin{array}{cccccccccccccccc}
\text{src1} & & & & & & & & & & & & & & & & \\
\text{src2} & & & & & & & & & & & & & & & & \\
\text{result} & & & & & & & & & & & & & & & & \\
\end{array}
\]
**Vector Shift Left Word VX-form**

vslw  VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>388</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 3

src1 ← VSR[VRA+32].word[i]

src2 ← VSR[VRB+32].word[i].bit[27:31]

VSR[VRT+32].word[i] ← src1 << src2
end

For each integer value i from 0 to 3, do the following.

Let src1 be the contents of word element i of VSR[VRA+32].

Let src2 be the contents of word element i of VSR[VRB+32].

src1 is shifted left by the number of bits specified in the low-order 5 bits of src2.

– Bits shifted out the most-significant bit are lost.

– Zeros are supplied to the vacated bits on the right.

The result is placed into word element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vslw

|------|---------------------|---------------------|---------------------|---------------------|

Register Data Layout for vsld

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
</tbody>
</table>
### Vector Shift Left Quadword VX-form

\[ \text{vslq} \quad \text{VRTX, VRAX, VRBX} \]

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>VRT</th>
<th>11</th>
<th>VRA</th>
<th>16</th>
<th>VRB</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

- if MSR.VEC=0 then Vector_Unavailable()
- \( VSR[VRT+32] \leftarrow VSR[VRA+32] \ll VSR[VRB+32].\text{bit}[57:63] \)

Let \( n \) be the contents of bits 57:63 of \( VSR[VRB+32] \).

Let \( \text{src1} \) be the contents of \( VSR[VRA+32] \).

Let \( \text{src2} \) be the contents of \( VSR[VRB+32] \).

\( \text{src1} \) is shifted left by the number of bits specified in the low-order 7 bits of \( \text{src2} \).

- Bits shifted out the most-significant bit are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into \( VSR[VRT+32] \).

**Special Registers Altered:**
None

**Register Data Layout for vslq**

<table>
<thead>
<tr>
<th>( \text{src1} )</th>
<th>( VSR[VRT+32] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{src2} )</td>
<td>( VSR[VRB+32] )</td>
</tr>
<tr>
<td>( \text{result} )</td>
<td>( VSR[VRT+32] )</td>
</tr>
</tbody>
</table>
6.9.6.2 Vector Integer Shift Right Instructions

**Vector Shift Right Byte VX-form**

vsrb  

\[
\begin{array}{cccccc}
4 & 5 & 6 & 9 & 15 & 516 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable()

\[
\text{do } i = 0 \text{ to 15} \\
\text{src1 } \leftarrow \text{VSR[VRA+32].byte[i]} \\
\text{src2 } \leftarrow \text{VSR[VRB+32].byte[i].bit[5:7]} \\
\text{VSR[VRT+32].byte[i]} \leftarrow \text{CHOP8(EXTZ(src1) \gg src2)} \\
\text{end}
\]

For each integer value \(i\) from 0 to 15, do the following. Let \(src1\) be the contents of byte element \(i\) of \(\text{VSR[VRA+32]}\).

Let \(src2\) be the contents of byte element \(i\) of \(\text{VSR[VRB+32]}\).

\(src1\) is shifted right by the number of bits specified in the low-order 3 bits of \(src2\).

- Bits shifted out the least-significant bit are lost.
- Zeros are supplied to the vacated bits on the left.

The result is placed into byte element \(i\) of \(\text{VSR[VRT+32]}\).

**Special Registers Altered:**

None

**Register Data Layout for vsrb**

\[
\begin{array}{cccccccccccccccccccc}
\text{src1} & \text{byte[0]} & \text{byte[1]} & \text{byte[2]} & \text{byte[3]} & \text{byte[4]} & \text{byte[5]} & \text{byte[6]} & \text{byte[7]} & \text{byte[8]} & \text{byte[9]} & \text{byte[10]} & \text{byte[11]} & \text{byte[12]} & \text{byte[13]} & \text{byte[14]} & \text{byte[15]} \\
\text{src2} & \text{byte[0]} & \text{byte[1]} & \text{byte[2]} & \text{byte[3]} & \text{byte[4]} & \text{byte[5]} & \text{byte[6]} & \text{byte[7]} & \text{byte[8]} & \text{byte[9]} & \text{byte[10]} & \text{byte[11]} & \text{byte[12]} & \text{byte[13]} & \text{byte[14]} & \text{byte[15]} \\
\text{result} & \text{byte[0]} & \text{byte[1]} & \text{byte[2]} & \text{byte[3]} & \text{byte[4]} & \text{byte[5]} & \text{byte[6]} & \text{byte[7]} & \text{byte[8]} & \text{byte[9]} & \text{byte[10]} & \text{byte[11]} & \text{byte[12]} & \text{byte[13]} & \text{byte[14]} & \text{byte[15]} \\
\end{array}
\]

**Vector Shift Right Halfword VX-form**

vsrh  

\[
\begin{array}{cccccc}
4 & 5 & 6 & 9 & 15 & 580 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable()

\[
\text{do } i = 0 \text{ to 7} \\
\text{src } \leftarrow \text{VSR[VRA+32].hword[i]} \\
\text{src2 } \leftarrow \text{VSR[VRB+32].hword[i].bit[12:15]} \\
\text{VSR[VRT+32].hword[i]} \leftarrow \text{CHOP16(EXTZ(src1) \gg src2)} \\
\text{end}
\]

For each integer value \(i\) from 0 to 7, do the following. Let \(src1\) be the contents of halfword element \(i\) of \(\text{VSR[VRA+32]}\).

Let \(src2\) be the contents of halfword element \(i\) of \(\text{VSR[VRB+32]}\).

\(src1\) is shifted right by the number of bits specified in the low-order 4 bits of \(src2\).

- Bits shifted out the least-significant bit are lost.
- Zeros are supplied to the vacated bits on the left.

The result is placed into halfword element \(i\) of \(\text{VSR[VRT+32]}\).

**Special Registers Altered:**

None

**Register Data Layout for vsrh**

\[
\begin{array}{cccccccccccccccccccc}
\text{src1} & \text{VSR[VRA+32].hword[0]} & \text{VSR[VRA+32].hword[1]} & \text{VSR[VRA+32].hword[2]} & \text{VSR[VRA+32].hword[3]} & \text{VSR[VRA+32].hword[4]} & \text{VSR[VRA+32].hword[5]} & \text{VSR[VRA+32].hword[6]} & \text{VSR[VRA+32].hword[7]} \\
\text{src2} & \text{VSR[VRB+32].hword[0]} & \text{VSR[VRB+32].hword[1]} & \text{VSR[VRB+32].hword[2]} & \text{VSR[VRB+32].hword[3]} & \text{VSR[VRB+32].hword[4]} & \text{VSR[VRB+32].hword[5]} & \text{VSR[VRB+32].hword[6]} & \text{VSR[VRB+32].hword[7]} \\
\text{result} & \text{VSR[VRT+32].hword[0]} & \text{VSR[VRT+32].hword[1]} & \text{VSR[VRT+32].hword[2]} & \text{VSR[VRT+32].hword[3]} & \text{VSR[VRT+32].hword[4]} & \text{VSR[VRT+32].hword[5]} & \text{VSR[VRT+32].hword[6]} & \text{VSR[VRT+32].hword[7]} \\
\end{array}
\]
Vector Shift Right Word VX-form

vsrw VRT, VRA, VRB

if MSR.VEC = 0 then Vector_Unavailable();

do i = 0 to 3

src1 ← VSR[VRA+32].word[i]
src2 ← VSR[VRB+32].word[i].bit[27:31]
VSR[VRT+32].word[i] ← CHOP32(EXTZ(src1) >> src2)
end

For each integer value i from 0 to 3, do the following.

Let src1 be the contents of word element i of VSR[VRA+32].

Let src2 be the contents of word element i of VSR[VRB+32].

src1 is shifted right by the number of bits specified in the low-order 5 bits of src2.

– Bits shifted out the least-significant bit are lost.

– Zeros are supplied to the vacated bits on the left.

The result is placed into word element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vsrw

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
<td>127</td>
</tr>
</tbody>
</table>

Vector Shift Right Doubleword VX-form

vsrd VRT, VRA, VRB

if MSR.VEC = 0 then Vector_Unavailable();

do i = 0 to 1

src1 ← VSR[VRA+32].dword[i]
src2 ← VSR[VRB+32].dword[i].bit[58:63]
VSR[VRT+32].dword[i] ← CHOP64(EXTZ(src1) >> src2)
end

For each integer value i from 0 to 1, do the following.

Let src1 be the contents of doubleword element i of VSR[VRA+32].

Let src2 be the contents of doubleword element i of VSR[VRB+32].

src1 are shifted right by the number of bits specified in bits 58:63 of src2.

– Bits shifted out the least-significant bit are lost.

– Zeros are supplied to the vacated bits on the left.

The result is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vsrd

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>
Vector Shift Right Quadword VX-form

vsrq VRT,VRA,VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>517</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();

src1 ← VSR[VRA+32]
src2 ← VSR[VRB+32].bit[57:63]
VSR[VRT+32] ← CHOP128(EXTZ(src1) >> src2)

Let src1 be the contents of VSR[VRA+32].
Let src2 be the contents of VSR[VRB+32].

src1 is shifted right by the number of bits specified in the low-order 7 bits of src2.

– Bits shifted out the least-significant bit are lost.
– Zeros are supplied to the vacated bits on the left.

The result is placed into VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vsrq

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
### 6.9.6.3 Vector Integer Shift Right Algebraic Instructions

#### Vector Shift Right Algebraic Byte VX-form

**vsrab**

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>772</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 15

src1 ← VSR[VRA+32].byte[i]

src2 ← VSR[VRB+32].byte[i].bit[5:7]

VSR[VRT+32].byte[i] ← CHOP8(EXTS(src1) >> src2)

end

For each integer value $i$ from 0 to 15, do the following.

Let **src1** be the contents of byte element $i$ of VSR[VRA+32].

Let **src2** be the contents of byte element $i$ of VSR[VRB+32].

**src1** is shifted right by the number of bits specified in the low-order 3 bits of **src2**.

- Bits shifted out the least-significant bit are lost.
- Copies of bit 0 of **src1** are supplied to the vacated bits on the left.

The result is placed into byte element $i$ of VSR[VRT+32].

#### Special Registers Altered:

None

**Register Data Layout for vsrab**

|-------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|

#### Vector Shift Right Algebraic Halfword VX-form

**vsrah**

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>836</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 7

src1 ← VSR[VRA+32].hword[i]

src2 ← VSR[VRB+32].hword[i].bit[12:15]

VSR[VRT+32].hword[i] ← CHOP16(EXTS(src1) >> src2)

end

For each integer value $i$ from 0 to 7, do the following.

Let **src1** be the contents of halfword element $i$ of VSR[VRA+32].

Let **src2** be the contents of halfword element $i$ of VSR[VRB+32].

**src1** is shifted right by the number of bits specified in the low-order 4 bits of **src2**.

- Bits shifted out the least-significant bit are lost.
- Copies of bit 0 of **src1** are supplied to the vacated bits on the left.

The result is placed into halfword element $i$ of VSR[VRT+32].

#### Special Registers Altered:

None

**Register Data Layout for vsrah**

|-------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
Vector Shift Right Algebraic Word VX-form

vsraw VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()  

for i = 0 to 3 

src1 ← VSR[VRA+32].word[i] 

src2 ← VSR[VRB+32].word[i].bit[27:31] 

VSR[VRT+32].word[i] ← CHOP32(EXTS(src1) >> src2)  

end

For each integer value i from 0 to 3, do the following.

Let src1 be the contents of word element i of VSR[VRA+32].

Let src2 be the contents of word element i of VSR[VRB+32].

src1 is shifted right by the number of bits specified in the low-order 5 bits of src2.

– Bits shifted out the least-significant bit are lost.

– Copies of bit 0 of src1 are supplied to the vacated bits on the left.

The result is placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vsraw

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
<td>127</td>
</tr>
</tbody>
</table>

Vector Shift Right Algebraic Doubleword VX-form

vsrad VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()  

for i = 0 to 1 

src1 ← VSR[VRA+32].dword[i] 

src2 ← VSR[VRB+32].dword[i].bit[58:63] 

VSR[VRT+32].dword[i] ← CHOP64(EXTS(src1) >> src2)  

end

For each integer value i from 0 to 1, do the following.

Let src1 be the contents of doubleword element i of VSR[VRA+32].

Let src2 be the contents of doubleword element i of VSR[VRB+32].

src1 is shifted right by the number of bits specified in the low-order 6 bits of src2.

– Bits shifted out the least-significant bit are lost.

– Copies of bit 0 of src1 are supplied to the vacated bits on the left.

The result is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vsrad

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>
Vector Shift Right Algebraic Quadword 
**VX-form**

vsraq VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable();

src1 ← VSR[VRA+32]
src2 ← VSR[VRB+32].bit[57:63]
VSR[VRT+32] ← CHOP128(EXTS(src1) >> src2)

Let src1 be the contents of VSR[VRA+32].
Let src2 be the contents of VSR[VRB+32].

src1 is shifted right by the number of bits specified in the low-order 7 bits of src2.

- Bits shifted out the least-significant bit are lost.
- Copies of bit 0 of src1 are supplied to the vacated bits on the left.

The result is placed into VSR[VRT+32].

**Special Registers Altered:**
None

**Register Data Layout for vsraq**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
6.10 Vector Floating-Point Instruction Set

6.10.1 Vector Floating-Point Arithmetic Instructions

**Vector Add Floating-Point VX-form**

vaddfp  VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
  src1 ← VSR[VRA+32].word[i]
  src2 ← VSR[VRB+32].word[i]
  VSR[VRT+32].word[i] ← bfp32_ADD(src1, src2)
end

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point value in word element i of VSR[VRA+32].

Let src2 be the single-precision floating-point value in word element i of VSR[VRB+32].

src1 is added to src2.

The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element i of VSR[VRT+32].

**Special Registers Altered:** None

**Register Data Layout for vaddfp & vsubfp**

|------|---------------------|---------------------|---------------------|---------------------|

| 0 | 32 | 64 | 96 | 127 |

**Vector Subtract Floating-Point VX-form**

vsubfp  VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>74</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
  src1 ← VSR[VRA+32].word[i]
  src2 ← VSR[VRB+32].word[i]
  VSR[VRT+32].word[i] ← bfp32_SUBTRACT(src1, src2)
end

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point value in word element i of VSR[VRA+32].

Let src2 be the single-precision floating-point value in word element i of VSR[VRB+32].

src2 is subtracted from src1.

The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element i of VSR[VRT+32].

**Special Registers Altered:** None

**Register Data Layout for vaddfp & vsubfp**

|------|---------------------|---------------------|---------------------|---------------------|

| 0 | 32 | 64 | 96 | 127 |
Vector Multiply-Add Floating-Point VA-form

**vmaddfp** VRT, VRA, VRC, VRB

```plaintext
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src1 ← VSR[VRA+32].word[i]
    src2 ← VSR[VRB+32].word[i]
    src3 ← VSR[VRC+32].word[i]
    result ← bfp32_MULTIPLY_ADD(src1, src3, src2)
    VSR[VRT+32].word[i] ← result
end
```

For each integer value $i$ from 0 to 3, do the following.

Let $src1$ be the single-precision floating-point value in word element $i$ of $VSR[VRA+32]$.

Let $src2$ be the single-precision floating-point value in word element $i$ of $VSR[VRB+32]$.

Let $src3$ be the single-precision floating-point value in word element $i$ of $VSR[VRC+32]$.

$src1$ is multiplied by $src3$.

$src2$ is added to the infinitely-precise product.

The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element $i$ of $VSR[VRT+32]$.

Special Registers Altered:
None

**Programming Note**

To use a multiply-add to perform an IEEE or Java compliant multiply, the addend must be $-0.0$. This is necessary to insure that the sign of a zero result will be correct when the product is $-0.0$ ($+0.0 + -0.0 = +0.0$, and $-0.0 + -0.0 = -0.0$). When the sign of a resulting $0.0$ is not important, then $+0.0$ can be used as an addend which may, in some cases, avoid the need for a second register to hold a $-0.0$ in addition to the integer floating-point $+0.0$ that may already be available.

Register Data Layout for vmaddfp & vnmsubfp

```
|------------|---------------------|---------------------|---------------------|---------------------|
```

Vector Negative Multiply-Subtract Floating-Point VA-form

**vnmsubfp** VRT, VRA, VRC, VRB

```plaintext
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src1 ← VSR[VRA+32].word[i]
    src2 ← VSR[VRB+32].word[i]
    src3 ← VSR[VRC+32].word[i]
    result ← bfp32_NEGATIVE_MULTIPLY_SUBTRACT(src1, src3, src2)
    VSR[VRT+32].word[i] ← result
end
```

For each integer value $i$ from 0 to 3, do the following.

Let $src1$ be the single-precision floating-point value in word element $i$ of $VSR[VRA+32]$.

Let $src2$ be the single-precision floating-point value in word element $i$ of $VSR[VRB+32]$.

Let $src3$ be the single-precision floating-point value in word element $i$ of $VSR[VRC+32]$.

$src1$ is multiplied by $src3$.

$src2$ is subtracted from the infinitely-precise product.

The intermediate result is rounded to the nearest single-precision floating-point number, then negated and placed into word element $i$ of $VSR[VRT+32]$.

Special Registers Altered:
None
6.10.2 Vector Floating-Point Maximum/Minimum Instructions

**Vector Maximum Floating-Point VX-form**

```plaintext
vmaxfp VRT, VRA, VRB
```

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src1 ← VSR[VRA+32].word[i]
    src2 ← VSR[VRB+32].word[i]
    VSR[VRT+32].word[i] ← bfp32_MAXIMUM(src1, src2)
end
```

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point value in word element i of VSR[VRA+32].

Let src2 be the single-precision floating-point value in word element i of VSR[VRB+32].

src1 is compared to src2.

The larger of the two values is placed into word element i of VSR[VRT+32].

The maximum of +0.0 and -0.0 is +0.0. The maximum of any value and a NaN is a QNaN.

**Special Registers Altered:**

None

**Register Data Layout for vmaxfp & vminfp**

|---------------|---------------------|---------------------|---------------------|---------------------|

```
4  VRT VRA VRB 1034
0  6  11  16  21  31
```

**Vector Minimum Floating-Point VX-form**

```plaintext
vminfp VRT, VRA, VRB
```

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src1 ← VSR[VRA+32].word[i]
    src2 ← VSR[VRB+32].word[i]
    VSR[VRT+32].word[i] ← bfp32_MINIMUM(src1, src2)
end
```

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point value in word element i of VSR[VRA+32].

Let src2 be the single-precision floating-point value in word element i of VSR[VRB+32].

src1 is compared to src2.

The smaller of the two values is placed into word element i of VSR[VRT+32].

The minimum of +0.0 and -0.0 is -0.0. The minimum of any value and a NaN is a QNaN.

**Special Registers Altered:**

None
6.10.3 Vector Floating-Point Rounding and Conversion Instructions

6.10.3.1 Vector Floating-Point Conversion Instructions

**Vector Convert with round to zero from floating-point To Signed Word format Saturate VX-form**

\[ \text{vctsxs} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>970</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3

src ← VSR[VRB+32].word[i]
VSR[VRT+32].word[i] ← sl32_CONVERT_FROM_BFP32(src, UIM)
end

For each integer value \( i \) from 0 to 3, do the following.

Let \( src \) be the signed fixed-point value in word element \( i \) of VSR[VRB+32].

\( src \) is multiplied by \( 2^{UIM} \). The product is converted to a 32-bit signed fixed-point integer using the rounding mode Round toward Zero.

- If the intermediate result is greater than \( 2^{31} - 1 \) the result saturates to \( 2^{31} - 1 \).
- If the intermediate result is less than \( -2^{31} \) the result saturates to \( -2^{31} \).

The result is placed into word element \( i \) of VSR[VRT+32].

**Special Registers Altered:**

SAT

**Extended Mnemonics:**

Example of an extended mnemonics for Vector Convert to Signed Fixed-Point Word Saturate:

Extended mnemonic: Equivalent to:

\[ \text{vcfpsxws VRT,VRB,UIM} \quad \text{vctsxs VRT,VRB,UIM} \]

**Register Data Layout for vctsxs & vctuxs**

|------|---------------------|---------------------|---------------------|---------------------|

---

**Vector Convert with round to zero from floating-point To Unsigned Word format Saturate VX-form**

\[ \text{vctuxs} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>906</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3

src ← VSR[VRB+32].word[i]
VSR[VRT+32].word[i] ← ui32_CONVERT_FROM_BFP32(src, UIM)
end

For each integer value \( i \) from 0 to 3, do the following.

Let \( src \) be the signed fixed-point value in word element \( i \) of VSR[VRB+32].

\( src \) is multiplied by \( 2^{UIM} \). The product is converted to a 32-bit unsigned fixed-point integer using the rounding mode Round toward Zero.

- If the intermediate result is greater than \( 2^{32} - 1 \) the result saturates to \( 2^{32} - 1 \).

The result is placed into word element \( i \) of VSR[VRT+32].

**Special Registers Altered:**

SAT

**Extended Mnemonics:**

Example of an extended mnemonics for Vector Convert to Unsigned Fixed-Point Word Saturate:

Extended mnemonic: Equivalent to:

\[ \text{vcfpuxws VRT,VRB,UIM} \quad \text{vctuxs VRT,VRB,UIM} \]
Vector Convert with round to nearest From Signed Word to floating-point format VX-form

vcfsx VRT,VRB,UIM

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>842</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 3
    src ← VSR[VRB+32].word[i]
    VSR[VRT+32].word[i] ← bfp32_CONVERT_FROM_SI32(src,UIM)
end

For each integer value i from 0 to 3, do the following.
Let src be the signed fixed-point value in word element i of VSR[VRB+32].

src is converted to the nearest single-precision floating-point value. Each result is divided by $2^{UIM}$ and placed into word element i of VSR[VRT+32].

Special Registers Altered:
None

Extended Mnemonics:
Examples of extended mnemonics for Vector Convert from Signed Fixed-Point Word:

Extended mnemonic: Equivalent to:
vcswfp VRT,VRB,UIM vcfsx VRT,VRB,UIM

Register Data Layout for vcfsx & vcfsx:

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
</tbody>
</table>

Vector Convert with round to nearest From Unsigned Word to floating-point format VX-form

vcfux VRT,VRB,UIM

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>778</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 3
    src ← VSR[VRB+32].word[i]
    VSR[VRT+32].word[i] ← bfp32_CONVERT_FROM_UI32(src,UIM)
end

For each integer value i from 0 to 3, do the following.
Let src be the unsigned fixed-point value in word element i of VSR[VRB+32].

src is converted to the nearest single-precision floating-point value. The result is divided by $2^{UIM}$ and placed into word element i of VSR[VRT+32].

Special Registers Altered:
None

Extended Mnemonics:
Examples of extended mnemonics for Vector Convert from Unsigned Fixed-Point Word:

Extended mnemonic: Equivalent to:
vcuxwfp VRT,VRB,UIM vcfux VRT,VRB,UIM

Register Data Layout for vcuxwfp & vcfxwfp:

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
</tbody>
</table>
6.10.3.2 Vector Floating-Point Round to Integral Instructions

**Vector Round to Floating-Point Integer toward -Infinity VX-form**

<table>
<thead>
<tr>
<th>vrfin</th>
<th>VRT, VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

- if MSR.VEC = 0 then Vector_Unavailable();
- do i = 0 to 3
  - src ← VSR[VRB+32].word[i]
  - VSR[VRT+32].word[i] ← bfp32_ROUND_TO_INTEGER_FLOOR(src)
- end

For each integer value i from 0 to 3, do the following.
Let src be the single-precision floating-point value in element i of VSR[VRB+32].
src is rounded to a single-precision floating-point integer using the rounding mode Round toward -Infinity.
The result is placed into the corresponding word element i of VSR[VRT+32].

**Special Registers Altered:**
None

**Register Data Layout for vrfin & vrfim**

|------|---------------------|---------------------|---------------------|---------------------|

**Programming Note**
The Vector Convert To Fixed-Point Word instructions support only the rounding mode Round toward Zero. A floating-point number can be converted to a fixed-point integer using any of the other three rounding modes by executing the appropriate Vector Round to Floating-Point Integer instruction before the Vector Convert To Fixed-Point Word instruction.

**Programming Note**
The fixed-point integers used by the Vector Convert instructions can be interpreted as consisting of 32-UIM integer bits followed by UIM fraction bits.
Vector Round to Floating-Point Integer toward +Infinity VX-form

vrflp VRT,VRB

if MSR.VEC=0 then Vector_Unavailable() if MSR.VEC=0 then Vector_Unavailable()
do i = 0 to 3 do i = 0 to 3
  src ← VSR[VRB+32].word[i] src ← VSR[VRB+32].word[i]
  VSR[VRT+32].word[i] ← bfp32_ROUND_TO_INTEGER_CEIL(src) VSR[VRT+32].word[i] ← bfp32_ROUND_TO_INTEGER_TRUNC(src)
end end

For each integer value i from 0 to 3, do the following.
Let src be the single-precision floating-point value in element i of VSR[VRB+32].

src is rounded to a single-precision floating-point integer using the rounding mode Round toward +Infinity.

The result is placed into the corresponding word element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vrflp & vrflz

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
</tr>
</tbody>
</table>

Vector Round to Floating-Point Integer toward Zero VX-form

vrflz VRT,VRB

if MSR.VEC=0 then Vector_Unavailable() if MSR.VEC=0 then Vector_Unavailable()
do i = 0 to 3 do i = 0 to 3
  src ← VSR[VRB+32].word[i] src ← VSR[VRB+32].word[i]
  VSR[VRT+32].word[i] ← bfp32_ROUND_TO_INTEGER_TRUNC(src) VSR[VRT+32].word[i] ← bfp32_ROUND_TO_INTEGER_Trunc(src)
end end

For each integer value i from 0 to 3, do the following.
Let src be the single-precision floating-point value in element i of VSR[VRB+32].

src is rounded to a single-precision floating-point integer using the rounding mode Round toward Zero.

The result is placed into the corresponding word element i of VSR[VRT+32].

Special Registers Altered:
None
6.10.4 Vector Floating-Point Compare Instructions

The Vector Floating-Point Compare instructions compare two VSRs word element by word element, interpreting the elements as single-precision floating-point numbers. With the exception of the Vector Compare Bounds Floating-Point instruction, they set the target VSR, and CR Field 6 if \( Rc = 1 \), in the same manner as do the Vector Integer Compare instructions; see Section 6.9.3.

The Vector Compare Bounds Floating-Point instruction sets the target VSR, and CR Field 6 if \( Rc = 1 \), to indicate whether the elements in \( VSR[VRA+32] \) are within the bounds specified by the corresponding element in \( VSR[VRB+32] \), as explained in the instruction description. A single-precision floating-point value \( x \) is said to be "within the bounds" specified by a single-precision floating-point value \( y \) if \(-y \leq x \leq y\).

### Vector Compare Bounds Floating-Point VC-form

<table>
<thead>
<tr>
<th>vcmbfp</th>
<th>VRT, VRA, VRB</th>
<th>((Rc = 0))</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcmbfp</td>
<td>VRT, VRA, VRB</td>
<td>((Rc = 1))</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 22 & 31 \\
\end{array}
\]

\[
\text{if MSR.VEC=0 then Vector_Unavailable()}
\]

\[
\text{do i = 0 to 3}
\]

\[
\begin{align*}
\text{src1} & \leftarrow VSR[VRA+32].\text{word}[i] \\
\text{src2} & \leftarrow VSR[VRB+32].\text{word}[i] \\
\text{le} & \leftarrow \text{bool\_COMPARE\_LE\_BFP32}(\text{src1}, \text{src2}) \\
\text{ge} & \leftarrow \text{bool\_COMPARE\_GE\_BFP32}(\text{src1}, \text{src2}) \\
VSR[VRT+32].\text{word}[i] & \leftarrow \text{~le} \oplus \text{~ge} \oplus 300
\end{align*}
\]

\[
\text{end}
\]

\[
\text{if Rc=1 then do}
\]

\[
\begin{align*}
\text{ib} & \leftarrow (\text{VSR[VRT+32]=0}) \\
\text{CR6} & \leftarrow 0b00 \oplus \text{ib} \oplus 0b10
\end{align*}
\]

\[
\text{end}
\]

For each integer value \( i \) from 0 to 3, do the following.

Let \( \text{src1} \) be the single-precision floating-point value in element \( i \) of \( VSR[VRA+32] \).

Let \( \text{src2} \) be the single-precision floating-point value in element \( i \) of \( VSR[VRB+32] \).

\( \text{src1} \) is compared to \( \text{src2} \).

A 2-bit value is formed that indicates whether \( \text{src1} \) is within the bounds specified by \( \text{src2} \), as follows.

- Bit 0 of the 2-bit value is set to 0 if \( \text{src1} \) is less than or equal to \( \text{src2} \), and is set to 1 otherwise.

- Bit 1 of the 2-bit value is set to 0 if \( \text{src1} \) is greater than or equal to the negation of \( \text{src2} \), and is set to 1 otherwise.

The 2-bit value is placed into the high-order two bits of word element \( i \) of \( VSR[VRT+32] \) and the remaining bits of element \( i \) are set to 0.

If \( Rc = 1 \), CR field 6 is set as follows.

**Bit Description**

0 Set to 0
1 Set to 0
2 Set to indicate whether all four elements in \( VSR[VRA+32] \) are within the bounds specified by the corresponding element in \( VSR[VRB+32] \), otherwise set to 0.
3 Set to 0

**Special Registers Altered:**

\( \text{CR field 6} \) ............................................... (if \( Rc = 1 \))

**Programming Note**

Each single-precision floating-point value in \( VSR[VRB+32] \) should be non-negative; if it is negative, the corresponding element in \( VSR[VRA+32] \) will necessarily be out of bounds.

One exception to this is when the value of an element in \( VSR[VRB+32] \) is -0.0 and the value of the corresponding element in \( VSR[VRA+32] \) is either +0.0 or -0.0 +0.0 and -0.0 compare equal to -0.0.

### Register Data Layout for vrflip & vrfiz

|------|---------------------|---------------------|---------------------|---------------------|

0 32 64 96 96 127
Vector Compare Equal Floating-Point VC-form

**vcmpqfp** VRT, VRA, VRB (Rc=0)

**vcmpqfp**. VRT, VRA, VRB (Rc=1)

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>%</th>
<th>198</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>22</td>
</tr>
</tbody>
</table>

> if MSR.VEC=0 then Vector_Unavailable()

all_true ← 1

all_false ← 1

do i = 0 to 3

src1 ← VSR[VRA+32].word[i]

src2 ← VSR[VRB+32].word[i]

if bool_COMPARE_EQ_BFP32(src1, src2) = 1 then

VSR[VRT+32].word[i] ← 0xFFFF_FFFF

else

VSR[VRT+32].word[i] ← 0x0000_0000

end

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point value in element i of VSR[VRA+32].

Let src2 be the single-precision floating-point value in element i of VSR[VRB+32].

src1 is compared to src2.

The contents of word element i of VSR[VRT+32] are set to all 1s if src1 is equal to src2, and are set to all 0s otherwise.

If src1 or src2 is a NaN, the contents of word element i of VSR[VRT+32] are set to all 0s, indicating "not equal to". If src1 and src2 are both infinity with the same sign, the contents of word element i of VSR[VRT+32] are set to all 1s, indicating "equal to".

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

CR field 6 ................................. (if Rc=1)

---

Register Data Layout for vcmdpfp[] & vcmpqfp[]

|------|---------------------|---------------------|---------------------|---------------------|

Vector Compare Greater Than or Equal Floating-Point VC-form

**vcmpefp** VRT, VRA, VRB (Rc=0)

**vcmpefp**. VRT, VRA, VRB (Rc=1)

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>%</th>
<th>454</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>22</td>
</tr>
</tbody>
</table>

> if MSR.VEC=0 then Vector_Unavailable()

all_true ← 1

all_false ← 1

do i = 0 to 3

src1 ← VSR[VRA+32].word[i]

src2 ← VSR[VRB+32].word[i]

if bool_COMPARE_GE_BFP32(src1, src2) = 1 then

VSR[VRT+32].word[i] ← 0xFFFF_FFFF

else

VSR[VRT+32].word[i] ← 0x0000_0000

end

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point value in element i of VSR[VRA+32].

Let src2 be the single-precision floating-point value in element i of VSR[VRB+32].

src1 is compared to src2.

The contents of word element i of VSR[VRT+32] are set to all 1s if src1 is greater than or equal to src2, and are set to all 0s otherwise.

If src1 or src2 is a NaN, the contents of word element i of VSR[VRT+32] are set to all 0s, indicating "not greater than or equal to". If src1 and src2 are both infinity with the same sign, the contents of word element i of VSR[VRT+32] are set to all 1s, indicating "greater than or equal to".

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

CR field 6 ................................. (if Rc=1)
Vector Compare Greater Than Floating-Point
VC-form

vcmpgtfp VRT, VRA, VRB (Rc=0)
vcmpgtfp VRT, VRA, VRB (Rc=1)

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>Rc</th>
<th>710</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

all_true ← 1
all_false ← 1
do i = 0 to 3
src1 ← VSR[VRA+32].word[i]
src2 ← VSR[VRB+32].word[i]
if bool_COMPARE_GT_BFP32(src1, src2) = 1 then
    VSR[VRT+32].word[i] ← 0xFFFF_FFFF
    all_false ← 0
else
    all_true ← 0
    VSR[VRT+32].word[i] ← 0x0000_0000
end
if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)

For each integer value i from 0 to 3, do the following.
Let src1 be the single-precision floating-point value in element i of VSR[VRA+32].
Let src2 be the single-precision floating-point value in element i of VSR[VRB+32].

src1 is compared to src2.
The contents of word element i of VSR[VRT+32] are set to all 1s if src1 is greater than src2, and are set to all 0s otherwise.

If src1 or src2 is a NaN, the contents of word element i of VSR[VRT+32] are set to all 0s, indicating "not greater than". If src1 and src2 are both infinity with the same sign, the contents of word element i of VSR[VRT+32] are set to all 0s, indicating "not greater than".

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 ......................... (if Rc=1)

Register Data Layout for vcmpgtfp[

|---------|---------------------|---------------------|---------------------|---------------------|

0 32 64 96 127

Chapter 6. Vector Facility 431
6.10.5 Vector Floating-Point Estimate Instructions

Vector 2 Raised to the Exponent Estimate Floating-Point VX-form

vexptefp VRT,VRB

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>394</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()]

do i = 0 to 3
    src ← VSR[VRB+32].word[i]
    VSR[VRT+32].word[i] ← bfp32_POWER2_ESTIMATE(src)
end

For each integer value i from 0 to 3, do the following.
The single-precision floating-point estimate of 2 raised to the power of single-precision floating-point element i of VSR[VRB+32] is placed into word element i of VSR[VRT+32].

Let x be any single-precision floating-point input value. Unless x < -146 or the single-precision floating-point result of computing 2 raised to the power x would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16. The most significant 12 bits of the estimate’s significand are monotonic. An integral input value returns an integral value when the result is representable.

The result for various special cases of the source value is given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>+0</td>
</tr>
<tr>
<td>-0</td>
<td>+1</td>
</tr>
<tr>
<td>+0</td>
<td>+1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Infinity</td>
</tr>
<tr>
<td>NaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

Special Registers Altered:
None

Register Data Layout for vexptefp

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
</tbody>
</table>

0 32 64 96 127
Vector Log Base 2 Estimate Floating-Point
VX-form

vlogefp VRT,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src ← VSR[VRB+32].word[i]
    VSR[VRT+32].word[i] ← bfp32_LOG_BASE2_ESTIMATE(src)
end

For each integer value i from 0 to 3, do the following.
The single-precision floating-point estimate of the base 2 logarithm of single-precision floating-point element i of VSR[VRB+32] is placed into the corresponding word element i of VSR[VRT+32].

Let x be any single-precision floating-point input value. Unless |x-1| is less than or equal to 0.125 or the single-precision floating-point result of computing the base 2 logarithm of x would be an infinity or a QNaN, the estimate has an absolute error in precision (absolute value of the difference between the estimate and the infinitely precise value) no greater than $2^{-5}$. Under the same conditions, the estimate has a relative error in precision no greater than one part in 8.

The most significant 12 bits of the estimate's significand are monotonic. The estimate is exact if $x=2^y$, where y is an integer between -149 and +127 inclusive. Otherwise the value placed into the element of VSR[VRT+32] may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Infinity</td>
<td>QNaN</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>QNaN</td>
</tr>
<tr>
<td>- 0</td>
<td>- Infinity</td>
</tr>
<tr>
<td>+0</td>
<td>- Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Infinity</td>
</tr>
<tr>
<td>NaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

Special Registers Altered:
None

Register Data Layout for vexptefp

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
<td>127</td>
</tr>
</tbody>
</table>
Vector Reciprocal Estimate Floating-Point VX-form

vrefp VRT,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src ← VSR[VRB+32].word[i]
    VSR[VRT+32].word[i] ← bfp32_RECIPROCAL_ESTIMATE(src)
end

For each integer value \( i \) from 0 to 3, do the following.
The single-precision floating-point estimate of the reciprocal of single-precision floating-point element \( i \) of \( VSR[VRB+32] \) is placed into word element \( i \) of \( VSR[VRT+32] \).

Unless the single-precision floating-point result of computing the reciprocal of a value would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 4096.

Note that results may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>-0</td>
</tr>
<tr>
<td>-0</td>
<td>-Infinity</td>
</tr>
<tr>
<td>+0</td>
<td>+Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+0</td>
</tr>
<tr>
<td>NaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

Special Registers Altered:

None

Register Data Layout for vrefp & vrsqrtefp

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
</table>

Vector Reciprocal Square Root Estimate Floating-Point VX-form

vrsqrtefp VRT,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
    src ← VSR[VRB+32].word[i]
    VSR[VRT+32].word[i] ← bfp32_RECIPROCAL_SQRT_ESTIMATE(src)
end

For each integer value \( i \) from 0 to 3, do the following.
The single-precision floating-point estimate of the reciprocal of the square root of single-precision floating-point element \( i \) of \( VSR[VRB+32] \) is placed into word element \( i \) of \( VSR[VRT+32] \).

Let \( x \) be any single-precision floating-point value. Unless the single-precision floating-point result of computing the reciprocal of the square root of \( x \) would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 4096.

Note that results may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>QNaN</td>
</tr>
<tr>
<td>&lt;0</td>
<td>QNaN</td>
</tr>
<tr>
<td>-0</td>
<td>-Infinity</td>
</tr>
<tr>
<td>+0</td>
<td>+Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+0</td>
</tr>
<tr>
<td>NaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

Special Registers Altered:

None
6.11 Vector Exclusive-OR-based Instructions

6.11.1 Vector AES Instructions

This section describes a set of instructions that support the Federal Information Processing Standards Publication 197 Advanced Encryption Standard for encryption and decryption.

**Vector AES Cipher VX-form**

```plaintext
vcipher VRT,VRA,VRB
```

1. If MSR.VEC=0 then Vector_Unavailable()
2. `State ← VSR[VRA+32]`
3. `RoundKey ← VSR[VRB+32]`
4. `vtemp1 ← SubBytes(State)`
5. `vtemp2 ← ShiftRows(vtemp1)`
6. `vtemp3 ← MixColumns(vtemp2)`
7. `VSR[VRT+32] ← vtemp3 ^ RoundKey`

Let `State` be the contents of `VSR[VRA+32]`, representing the intermediate state array during AES cipher operation.

Let `RoundKey` be the contents of `VSR[VRB+32]`, representing the round key.

One round of an AES cipher operation is performed on the intermediate `State` array, sequentially applying the transforms, `SubBytes()`, `ShiftRows()`, `MixColumns()`, and `AddRoundKey()`, as defined in FIPS-197.

The result is placed into `VSR[VRT+32]`, representing the new intermediate state of the cipher operation.

**Special Registers Altered:**

None

**Register Data Layout for vcipher & vcipherlast**

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1288</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>output</td>
<td>21</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src2</td>
<td>VSR[VRA+32]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Vector AES Cipher Last VX-form**

```plaintext
vcipherlast VRT,VRA,VRB
```

1. If MSR.VEC=0 then Vector_Unavailable()
2. `State ← VSR[VRA+32]`
3. `RoundKey ← VSR[VRB+32]`
4. `vtemp1 ← SubBytes(State)`
5. `vtemp2 ← ShiftRows(vtemp1)`
6. `VSR[VRT+32] ← vtemp2 ^ RoundKey`

Let `State` be the contents of `VSR[VRA+32]`, representing the intermediate state array during AES cipher operation.

Let `RoundKey` be the contents of `VSR[VRB+32]`, representing the round key.

The final round in an AES cipher operation is performed on the intermediate `State` array, sequentially applying the transforms, `SubBytes()`, `ShiftRows()`, `AddRoundKey()`, as defined in FIPS-197.

The result is placed into `VSR[VRT+32]`, representing the final state of the cipher operation.

**Special Registers Altered:**

None
Vector AES Inverse Cipher VX-form

\[
\text{vncipher} \quad \text{VRT, VRA, VRB}
\]

Vector AES Inverse Cipher Last VX-form

\[
\text{vncipherlast} \quad \text{VRT, VRA, VRB}
\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

State $\leftarrow$ VSR[VRA+32]
RoundKey $\leftarrow$ VSR[VRB+32]
vtemp1 $\leftarrow$ InvShiftRows(State)
vtemp2 $\leftarrow$ InvSubBytes(vtemp1)
vtemp3 $\leftarrow$ vtemp2 $^\wedge$ RoundKey
VSR[VRT+32] $\leftarrow$ InvMixColumns(vtemp3)

Let State be the contents of VSR[VRA+32], representing the intermediate state array during AES inverse cipher operation.

Let RoundKey be the contents of VSR[VRB+32], representing the round key.

One round of an AES inverse cipher operation is performed on the intermediate State array, sequentially applying the transforms, InvShiftRows(), InvSubBytes(), AddRoundKey(), and InvMixColumns(), as defined in FIPS-197.

The result is placed into VSR[VRT+32], representing the new intermediate state of the inverse cipher operation.

Special Registers Altered:
None

Register Data Layout for vncipher & vncipherlast

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

State $\leftarrow$ VSR[VRA+32]
RoundKey $\leftarrow$ VSR[VRB+32]
vtemp1 $\leftarrow$ InvShiftRows(State)
vtemp2 $\leftarrow$ InvSubBytes(vtemp1)
VSR[VRT+32] $\leftarrow$ vtemp2 $^\wedge$ RoundKey

Let State be the contents of VSR[VRA+32], representing the intermediate state array during AES inverse cipher operation.

Let RoundKey be the contents of VSR[VRB+32], representing the round key.

The final round in an AES inverse cipher operation is performed on the intermediate State array, sequentially applying the transforms, InvShiftRows(), InvSubBytes(), and AddRoundKey(), as defined in FIPS-197.

The result is placed into VSR[VRT+32], representing the final state of the inverse cipher operation.

Special Registers Altered:
None
Vector AES SubBytes VX-form

vsbox VRT, VRA

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>11</td>
<td>16</td>
<td>1480</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

State ← VSR[VRA+32]
VSR[VRT+32] ← SubBytes(State)

Let State be the contents of VSR[VRA+32], representing the intermediate state array during AES cipher operation.

The result of applying the transform, SubBytes(), on State, as defined in FIPS-197, is placed into VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vsbox

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>VSR[VRA+32]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.11.2 Vector SHA-256 and SHA-512 Sigma Instructions

This section describes a set of instructions that support the Federal Information Processing Standards Publication 180-3 Secure Hash Standard.

Vector SHA-512 Sigma Doubleword VX-form

For each integer value \(i\) from 0 to 1, do the following.

When \(ST=0\) and bit \(2 \times i\) of \(SI\) is 0, a SHA-512 \(\sigma_0\) function is performed on the contents of doubleword element \(i\) of \(VSR[VRA+32]\) and the result is placed into doubleword element \(i\) of \(VSR[VRT+32]\).

When \(ST=0\) and bit \(2 \times i\) of \(SI\) is 1, a SHA-512 \(\sigma_1\) function is performed on the contents of doubleword element \(i\) of \(VSR[VRA+32]\) and the result is placed into doubleword element \(i\) of \(VSR[VRT+32]\).

When \(ST=1\) and bit \(2 \times i\) of \(SI\) is 0, a SHA-512 \(\Sigma_0\) function is performed on the contents of doubleword element \(i\) of \(VSR[VRA+32]\) and the result is placed into doubleword element \(i\) of \(VSR[VRT+32]\).

When \(ST=1\) and bit \(2 \times i\) of \(SI\) is 1, a SHA-512 \(\Sigma_1\) function is performed on the contents of doubleword element \(i\) of \(VSR[VRA+32]\) and the result is placed into doubleword element \(i\) of \(VSR[VRT+32]\).

Bits 1 and 3 of \(SI\) are reserved.

Special Registers Altered:

None

Register Data Layout for vshasigmad

\[
\begin{array}{cccccc}
\text{src1} & \text{VSR[VRA+32].dword[0]} & \text{VSR[VRA+32].dword[1]} \\
\text{src2} & \text{VSR[VRB+32].dword[0]} & \text{VSR[VRB+32].dword[1]} \\
\text{result} & \text{VSR[VRT+32].dword[0]} & \text{VSR[VRT+32].dword[1]} \\
\end{array}
\]
Vector SHA-256 Sigma Word VX-form

vshasigmaw VRT,VRA,ST,SIX

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>ST</th>
<th>SIX</th>
<th>1666</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>11</td>
<td>10</td>
<td>17</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

doi = 0 to 3

src ← VSR[VRA+32].word[i]

if ST=0 & SIX.bit[i]=0 then // SHA-256 σ0 function
VSR[VRT+32].word[i] ← (src >>> 7) ^
VSR[VRT+32].word[i] ← (src >>> 18) ^
VSR[VRT+32].word[i] ← (src >> 3)

if ST=0 & SIX.bit[i]=1 then // SHA-256 σ1 function
VSR[VRT+32].word[i] ← (src >>> 17) ^
VSR[VRT+32].word[i] ← (src >>> 19) ^
VSR[VRT+32].word[i] ← (src >>> 10)

if ST=1 & SIX.bit[i]=0 then // SHA-256 Σ0 function
VSR[VRT+32].word[i] ← (src >>> 2) ^
VSR[VRT+32].word[i] ← (src >>> 13) ^
VSR[VRT+32].word[i] ← (src >>> 22)

if ST=1 & SIX.bit[i]=1 then // SHA-256 Σ1 function
VSR[VRT+32].word[i] ← (src >>> 6) ^
VSR[VRT+32].word[i] ← (src >>> 11) ^
VSR[VRT+32].word[i] ← (src >>> 25)

end

For each integer value i from 0 to 3, do the following.

When ST=0 and bit i of SIX is 0, a SHA-256 σ0 function is performed on the contents of word element i of VSR[VRA+32] and the result is placed into word element i of VSR[VRT+32].

When ST=0 and bit i of SIX is 1, a SHA-256 σ1 function is performed on the contents of word element i of VSR[VRA+32] and the result is placed into word element i of VSR[VRT+32].

When ST=1 and bit i of SIX is 0, a SHA-256 Σ0 function is performed on the contents of word element i of VSR[VRA+32] and the result is placed into word element i of VSR[VRT+32].

When ST=1 and bit i of SIX is 1, a SHA-256 Σ1 function is performed on the contents of word element i of VSR[VRA+32] and the result is placed into word element i of VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vshasigmaw

|---|---|---|---|---|
6.11.3 Vector Binary Polynomial Multiplication Instructions

This section describes a set of binary polynomial multiply-sum instructions. Corresponding elements are multiplied and the exclusive-OR of each even-odd pair of products sum, useful for a variety of finite field arithmetic operations.

**Vector Polynomial Multiply-Sum Byte VX-form**

**vpmsumb**  
VRT,VRA,VRB

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

```
if MSR.VEC=0 then Vector_Unavailable() |

do i = 0 to 15
   prod[i].bit[0:14] ← 0
   srcA ← VSR[VRA+32].byte[i]
   srcB ← VSR[VRB+32].byte[i]
   do j = 0 to 7
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   do j = 8 to 14
      do k = j-7 to 7
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   do i = 0 to 7
      VSR[VRT+32].hword[i] ← 0b0 || (prod[2×i] ^ prod[2×i+1])
   end
```

For each integer value \( i \) from 0 to 15, do the following.
Let \( \text{prod}[i] \) be the 15-bit result of a binary polynomial multiplication of the contents of byte element \( i \) of \( \text{VSR}[\text{VRA}+32] \) and the contents of byte element \( i \) of \( \text{VSR}[\text{VRB}+32] \).

For each integer value \( i \) from 0 to 7, do the following.
The exclusive-OR of \( \text{prod}[2\times i] \) and \( \text{prod}[2\times i+1] \) is placed in bits 1:15 of halfword element \( i \) of \( \text{VSR}[\text{VRT}+32] \). Bit 0 of halfword element \( i \) of \( \text{VSR}[\text{VRT}+32] \) is set to 0.

**Special Registers Altered:**
None

**Register Data Layout for vpmsumb**

```
```

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127
Vector Polynomial Multiply-Sum Halfword VX-form

vpmsumh VRT,VRA,VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   prod.bit[0:30] ← 0
   srcA ← VSR[VRA+32].halfword[i]
   srcB ← VSR[VRB+32].halfword[i]
   do j = 0 to 15
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   do j = 16 to 30
      do k = j-15 to 15
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   VSR[VRT+32].word[i] ← 0b0 || (prod[i] ^ prod[i+1])

For each integer value i from 0 to 7, do the following.
Let prod[i] be the 31-bit result of a binary polynomial multiplication of the contents of halfword element i of VSR[VRA+32] and the contents of halfword element i of VSR[VRB+32].

For each integer value i from 0 to 3, do the following.
The exclusive-OR of prod[2*i] and prod[2*i+1] is placed in bits 1:31 of word element i of VSR[VRT+32]. Bit 0 of word element i of VSR[VRT+32] is set to 0.

Special Registers Altered:
None

Register Data Layout for vpmsumh

|------|----------------------|----------------------|----------------------|----------------------|

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
<td>64</td>
</tr>
</tbody>
</table>
Vector Polynomial Multiply-Sum Word VX-form

If MSR.VEC=0 then Vector_Unavailable() if MSR.VEC=0 then Vector_Unavailable()

\[
\text{prod}[i].\text{bit}[0:62] \leftarrow 0
\]

\[
\text{src}A \leftarrow \text{VSR}[\text{VRA}+32].\text{word}[i]
\]

\[
\text{src}B \leftarrow \text{VSR}[\text{VRB}+32].\text{word}[i]
\]

\[
do i = 0 to 3
do j = 0 to 31
\]

\[
gbit \leftarrow \text{src}A.\text{bit} [k] \& \text{src}B.\text{bit}[j-k]
gbit \leftarrow \text{src}A.\text{bit} [k] \& \text{src}B.\text{bit}[j-k]
\]

\[
\text{prod}[i].\text{bit}[j] \leftarrow \text{prod}[i].\text{bit}[j] ^ \text{gbit}
\]

\[
end
\]

\[
end
\]

\[
do j = 32 to 62
\]

\[
do k = j-31 to 31
\]

\[
gbit \leftarrow (\text{src}A.\text{bit} [k] \& \text{src}B.\text{bit}[j-k])
gbit \leftarrow (\text{src}A.\text{bit} [k] \& \text{src}B.\text{bit}[j-k])
\]

\[
\text{prod}[i].\text{bit}[j] \leftarrow \text{prod}[i].\text{bit}[j] ^ \text{gbit}
\]

\[
end
\]

\[
end
\]

\[
\text{VSR}[\text{VRT}+32].\text{dword}[0] \leftarrow 0b0 \| (\text{prod}[0] ^ \text{prod}[1])
\]

\[
\text{VSR}[\text{VRT}+32].\text{dword}[1] \leftarrow 0b0 \| (\text{prod}[2] ^ \text{prod}[3])
\]

For each integer value \(i\) from 0 to 3, do the following.

Let \(\text{prod}[i]\) be the 63-bit result of a binary polynomial multiplication of the contents of word element \(i\) of \(\text{VSR}[\text{VRA}+32]\) and the contents of word element \(i\) of \(\text{VSR}[\text{VRB}+32]\).

For each integer value \(i\) from 0 to 1, do the following.

The exclusive-OR of \(\text{prod}[2\times i]\) and \(\text{prod}[2\times i+1]\) is placed in bits 1:63 of doubleword element \(i\) of \(\text{VSR}[\text{VRT}+32]\). Bit 0 of doubleword element \(i\) of \(\text{VSR}[\text{VRT}+32]\) is set to 0.

**Special Registers Altered:**

None

**Register Data Layout for \text{vpmsumw}\)**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
<td>127</td>
</tr>
</tbody>
</table>
Vector Polynomial Multiply-Sum Doubleword VX-form

vpmsumd VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
  prod[i].bit[0:126] ← 0
  srcA ← VSR[VRA+32].dword[i]
  srcB ← VSR[VRB+32].dword[i]
  do j = 0 to 63
    do k = 0 to j
      gbit ← srcA.bit[k] & srcB.bit[j-k]
      prod[i].bit[j] ← prod[i].bit[j] ^ gbit
    end
  end
  do j = 64 to 126
    do k = j-63 to 63
      gbit ← srcA.bit[k] & srcB.bit[j-k]
      prod[i].bit[j] ← prod[i].bit[j] ^ gbit
    end
  end
  VSR[VRT+32] ← 0b0 || (prod[0] ^ prod[1])

Let prod[0] be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 0 of VSR[VRA+32] and the contents of doubleword element 0 of VSR[VRB+32].

Let prod[1] be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 1 of VSR[VRA+32] and the contents of doubleword element 1 of VSR[VRB+32].

The exclusive-OR of prod[0] and prod[1] is placed in bits 1:127 of VSR[VRT+32]. Bit 0 of VSR[VRT+32] is set to 0.

Special Registers Altered:

None

Register Data Layout for vpmsumd

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>
6.11.4 Vector Permute & Exclusive-OR Instruction

**Vector Permute & Exclusive-OR VA-form**

```
vpermxor  VRT,VRA,VRB,VRC
```

```plaintext
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 15
    indexA ← VSR[VRC+32].byte[i].bit[0:3]
    indexB ← VSR[VRC+32].byte[i].bit[4:7]
    src1 ← VSR[VRA+32].byte[indexA]
    src2 ← VSR[VRB+32].byte[indexB]
    VSR[VRT+32].byte[i] ← src1 ^ src2
end
```

For each integer value `i` from 0 to 15, do the following.
Let `indexA` be the contents of bits 0:3 of byte element `i` of VSR[VRC+32].
Let `indexB` be the contents of bits 4:7 of byte element `i` of VSR[VRC+32].
The exclusive OR of the contents of byte element `indexA` of VSR[VRA+32] and the contents of byte element `indexB` of VSR[VRB+32] is placed into byte element `i` of VSR[VRT+32].

**Special Registers Altered:**
None

**Register Data Layout for vpermxor**

```
```

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127
6.12 Vector Bit Manipulation Instructions

6.12.1 Vector Gather Bits Instructions

*Vector Gather Bits by Bytes by Doubleword VX-form*

Let \( s_r c \) be the contents of \( VSR[VRB+32] \), composed of two doubleword elements numbered 0 and 1.

Let each doubleword element be composed of eight bytes numbered 0 through 7.

An 8-bit \( \times \) 8-bit bit-matrix transpose is performed on the contents of each doubleword element of \( VSR[VRB+32] \) (see Figure 104).

For each integer value \( i \) from 0 to 1, do the following,

1. The contents of bit 0 of each byte of doubleword element \( i \) of \( VSR[VRB+32] \) are concatenated and placed into byte 0 of doubleword element \( i \) of \( VSR[VRT+32] \).
2. The contents of bit 1 of each byte of doubleword element \( i \) of \( VSR[VRB+32] \) are concatenated and placed into byte 1 of doubleword element \( i \) of \( VSR[VRT+32] \).
3. The contents of bit 2 of each byte of doubleword element \( i \) of \( VSR[VRB+32] \) are concatenated and placed into byte 2 of doubleword element \( i \) of \( VSR[VRT+32] \).
4. The contents of bit 3 of each byte of doubleword element \( i \) of \( VSR[VRB+32] \) are concatenated and placed into byte 3 of doubleword element \( i \) of \( VSR[VRT+32] \).
5. The contents of bit 4 of each byte of doubleword element \( i \) of \( VSR[VRB+32] \) are concatenated and placed into byte 4 of doubleword element \( i \) of \( VSR[VRT+32] \).
6. The contents of bit 5 of each byte of doubleword element \( i \) of \( VSR[VRB+32] \) are concatenated and placed into byte 5 of doubleword element \( i \) of \( VSR[VRT+32] \).
7. The contents of bit 6 of each byte of doubleword element \( i \) of \( VSR[VRB+32] \) are concatenated and placed into byte 6 of doubleword element \( i \) of \( VSR[VRT+32] \).
8. The contents of bit 7 of each byte of doubleword element \( i \) of \( VSR[VRB+32] \) are concatenated and placed into byte 7 of doubleword element \( i \) of \( VSR[VRT+32] \).

**Special Registers Altered:**

None

---

**Figure 104. Vector Gather Bits by Bytes by Doubleword**

**Register Data Layout for vgbbd**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32].dword[0]</th>
<th>VSR[VRB+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector Gather every Nth Bit VX-form

\[ \text{vgnb RT,VRB,N} \]

\[
\begin{array}{cccccc}
0 & 6 & 11 & 13 & 16 & 21 & 1228 \\
\end{array}
\]

- If MSR.VEC=0 then Vector_Unavailable()
- If N<2 \text{ or } N>7 then result ← UNDEFINED
- Else do
  - \( j \leftarrow 0 \)
  - \( \text{result} \leftarrow 0x0000_0000_0000_0000 \)
  - Do \( i = 0 \) to \( 127 \) by \( N \)
     \quad \text{result.bit}[j] \leftarrow \text{VSR}[VRB+32].bit[i]
     \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad j \leftarrow j + 1
  - End
- End
  - \( \text{GPR}[RT] \leftarrow \text{result} \)

Starting with bit 0, the contents of every \( N^{\text{th}} \) bit of \( \text{VSR}[VRB+32] \) are concatenated and placed into \( \text{GPR}[RT] \).

\( N \) can be any value between 2 and 7, inclusive.

**Special Registers Altered:**
- None

**Register Data Layout for vpmsumd**

- \( \text{src} \)
- \( \text{VSR}[VRB+32] \)
- \( \text{result} \)
- \( \text{GPR}[RT] \)
6.12.2 Vector Count Leading Zeros Instructions

**Vector Count Leading Zeros Byte VX-form**

```
vclzb VRT,VRB
```

If MSR.VEC=0 then Vector_Unavailable()

```
do i = 0 to 15
    n ← 0
    do while n < 8
        if VSR[VRB+32].byte[i].bit[n] = 0b1 then leave
        n ← n + 1
    end
    VSR[VRT+32].byte[i] ← n
end
```

For each integer value \( i \) from 0 to 15, do the following:
A count of the number of consecutive zero bits starting at bit 0 of byte element \( i \) of VSR[VRB+32] is placed into byte element \( i \) of VSR[VRT+32]. This number ranges from 0 to 8, inclusive.

**Special Registers Altered:**
None

**Register Data Layout for vclzb**

|---------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
```

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127
```

**Vector Count Leading Zeros Halfword VX-form**

```
vclzh VRT,VRB
```

If MSR.VEC=0 then Vector_Unavailable()

```
do i = 0 to 7
    n ← 0
    do while n < 16
        if VSR[VRB+32].hword[i].bit[n] = 0b1 then leave
        n ← n + 1
    end
    VSR[VRT+32].hword[i] ← n
end
```

For each integer value \( i \) from 0 to 7, do the following:
A count of the number of consecutive zero bits starting at bit 0 of halfword element \( i \) of VSR[VRB+32] is placed into halfword element \( i \) of VSR[VRT+32]. This number ranges from 0 to 16, inclusive.

**Special Registers Altered:**
None

**Register Data Layout for vclzh**

|-------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
```

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127
Vector Count Leading Zeros Word VX-form

vclzw       VRT,VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1922</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable();

do i = 0 to 3
n ← 0
do while n < 32
  if VSR[VRB+32].word[i].bit[n] = 0b1 then leave
  n ← n + 1
end
VSR[VRT+32].word[i] ← n
end

For each integer value \( i \) from 0 to 3, do the following.
A count of the number of consecutive zero bits starting at bit 0 of word element \( i \) of VSR[VRB+32]
is placed into word element \( i \) of VSR[VRT+32]. This number ranges from 0 to 32, inclusive.

Special Registers Altered:
None

Register Data Layout for vclzw

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
<td>127</td>
</tr>
</tbody>
</table>
**Vector Count Leading Zeros Doubleword VX-form**

**vclzd**  
VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1986</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
  n ← 0
  do while (n<64) & (VSR[VRB+32].dword[i].bit[n]=0b0)
    n ← n + 1
  end
  VSR[VRT+32].dword[i] ← n
end

For each integer value \(i\) from 0 to 1, do the following. A count of the number of consecutive zero bits starting at bit 0 of doubleword element \(i\) of VSR[VRB+32] is placed into doubleword element \(i\) of VSR[VRT+32]. This number ranges from 0 to 64, inclusive.

**Special Registers Altered:**  
None

---

**Vector Count Leading Zeros Doubleword under bit Mask VX-form**

**vclzdm**  
VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
  count ← 0
  do j = 0 to 63
    if VSR[VRB+32].dword[i].bit[j]=1 then do
      if VSR[VRA+32].dword[i].bit[j]=1 then break
      count ← count + 1
    end
  end
  VSR[VRT+32].dword[i] ← EXTZ64(count)
end

For each integer value \(i\) from 0 to 1, starting on the left, count the number of consecutive 0 bits in doubleword element \(i\) of VSR[VRA+32] corresponding to 1 bits in doubleword element \(i\) of VSR[VRB+32]. Place count in doubleword element \(i\) of VSR[VRT+32].

**Special Registers Altered:**  
None

---

**Register Data Layout for vclzd**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32].dword[0]</th>
<th>VSR[VRB+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>VSR[VRT+32].dword[0]</th>
<th>VSR[VRT+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Register Data Layout for vclzdm**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32].dword[0]</th>
<th>VSR[VRA+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2</th>
<th>VSR[VRB+32].dword[0]</th>
<th>VSR[VRB+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>VSR[VRT+32].dword[0]</th>
<th>VSR[VRT+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 6.12.3 Vector Count Trailing Zeros Instructions

#### Vector Count Trailing Zeros Byte VX-form

**vctzb**  
\[
\begin{array}{ccccccc}
\text{VRT} & \text{VRB} \\
\hline
0 & 4 & 6 & 11 & 16 & 21 & 1538 \\
\end{array}
\]

- If MSR.VEC=0 then Vector_Unavailable()
- \(i = 0\) to \(15\)
- \(n \leftarrow 0\)
- \(\text{do while } n < 8\)
  - \(\text{if } \text{VSR}[VRB+32].\text{byte}[i].\text{bit}[7-n] = 0b1 \text{ then leave}\)
  - \(n \leftarrow n + 1\)
- \(\text{VSR}[VRT+32].\text{byte}[i] \leftarrow \text{CHOP}(\text{EXTZ}(n))\)

For each integer value \(i\) from 0 to 15, do the following.

A count of the number of consecutive zero bits starting at bit 7 of byte element \(i\) of VSR[VRB+32] is placed into byte element \(i\) of VSR[VRT+32]. This number ranges from 0 to 8, inclusive.

**Special Registers Altered:** None

**Register Data Layout for vctzb**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
</tr>
</tbody>
</table>

#### Vector Count Trailing Zeros Halfword VX-form

**vctzh**  
\[
\begin{array}{ccccccc}
\text{VRT} & \text{VRB} \\
\hline
0 & 4 & 29 & 16 & 21 & 1538 \\
\end{array}
\]

- If MSR.VEC=0 then Vector_Unavailable()
- \(i = 0\) to \(7\)
- \(n \leftarrow 0\)
- \(\text{do while } n < 16\)
  - \(\text{if } \text{VSR}[VRB+32].\text{hword}[i].\text{bit}[15-n] = 0b1 \text{ then leave}\)
  - \(n \leftarrow n + 1\)
- \(\text{VSR}[VRT+32].\text{hword}[i] \leftarrow \text{CHOP16}(\text{EXTZ}(n))\)

For each integer value \(i\) from 0 to 7, do the following.

A count of the number of consecutive zero bits starting at bit 15 of halfword element \(i\) of VSR[VRB+32] is placed into halfword element \(i\) of VSR[VRT+32]. This number ranges from 0 to 16, inclusive.

**Special Registers Altered:** None

**Register Data Layout for vctzh**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>63</td>
<td>24</td>
<td>86</td>
<td>48</td>
<td>09</td>
<td>61</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>63</td>
<td>24</td>
<td>86</td>
<td>48</td>
<td>09</td>
<td>61</td>
</tr>
</tbody>
</table>
**Vector Count Trailing Zeros Word VX-form**

```plaintext
vctzw VRT,VRB
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>30</th>
<th>VRB</th>
<th>1538</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

*if MSR.VEC=0 then Vector_Unavailable*;

```plaintext
do i = 0 to 3
    n ← 0
    do while n < 32
        if VSR[VRB+32].word[i].bit[31-n] = 0b1 then leave
        n ← n + 1
    end
    VSR[VRT+32].word[i] ← CHOP32(EXTZ(n))
end
```

For each integer value \(i\) from 0 to 3, do the following.

A count of the number of consecutive zero bits starting at bit 31 of word element \(i\) of VSR[VRB+32] is placed into word element \(i\) of VSR[VRT+32]. This number ranges from 0 to 32, inclusive.

**Special Registers Altered:**

None

---

**Register Data Layout for vctzw**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
<td>127</td>
</tr>
</tbody>
</table>
Vector Count Trailing Zeros Doubleword VX-form

\[
vctzd \quad VRT, VRB
\]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1538</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{do } i & = 0 \text{ to } 1 \\
\text{n} & \leftarrow 0 \\
\text{do while } n < 64 \\
\text{if VSR[VRB+32].dword[i].bit[63-n] = 0b1 then leave} \\
\text{n} & \leftarrow n + 1 \\
\text{end} \\
\text{VSR[VRT+32].dword[i] \leftarrow CHOP64(EXTZ(n))} \\
\text{end}
\end{align*}
\]

For each integer value \(i\) from 0 to 1, do the following. A count of the number of consecutive zero bits starting at bit 63 of doubleword element \(i\) of \(\text{VSR[VRB+32]}\) is placed into doubleword element \(i\) of \(\text{VSR[VRT+32]}\). This number ranges from 0 to 64, inclusive.

Special Registers Altered:
None

Register Data Layout for \(vctzd\)

<table>
<thead>
<tr>
<th>src</th>
<th>dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>

Vector Count Trailing Zeros Doubleword under bit Mask VX-form

\[
vctzd \quad VRT, VRA, VRB
\]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1988</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{do } i & = 0 \text{ to } 1 \\
\text{count} & \leftarrow 0 \\
\text{do } j & = 0 \text{ to } 63 \\
\text{if VSR[VRB+32].dword[i].bit[63-j] = 1 then do} \\
\text{if VSR[VRA+32].dword[i].bit[63-j] = 1 then break} \\
\text{count} & \leftarrow count + 1 \\
\text{end} \\
\text{end} \\
\text{VSR[VRT+32].dword[i] \leftarrow EXTZ64(count)} \\
\text{end}
\end{align*}
\]

For each integer value \(i\) from 0 to 1, starting on the right, count the number of consecutive 0 bits in doubleword element \(i\) of \(\text{VSR[VRA+32]}\) corresponding to 1 bits in doubleword element \(i\) of \(\text{VSR[VRB+32]}\). Place count in doubleword element \(i\) of \(\text{VSR[VRT+32]}\).

Special Registers Altered:
None

Register Data Layout for \(vctzd\)

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRA+32].dword[0]</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>

Register Data Layout for \(vctzdm\)

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRA+32].dword[0]</td>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>
6.12.4 Vector Count Leading/Trailing Zero LSB Instructions

**Vector Count Leading Zero Least-Significant Bits Byte VX-form**

```vhdl
vclzlsbb RT,VRB
```

if MSR.VEC=0 then Vector_Unavailable()

```vhdl
count ← 0
do while count < 16
  count ← count + 1
end
GPR[RT] ← EXTZ64(count)
```

Let `count` be the number of contiguous leading byte elements in `VSR[VRB+32]` having a zero least-significant bit. `count` is placed into GPR[RT].

**Special Registers Altered:**

None

**Register Data Layout for vclzlsbb & vctzlsbb**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 8 16 24 32 40 48 56</td>
<td>64 72 80 88 96 104 112 120 127</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result</td>
<td>GPR[RT]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Vector Count Trailing Zero Least-Significant Bits Byte VX-form**

```vhdl
vctzlsbb RT,VRB
```

if MSR.VEC=0 then Vector_Unavailable()

```vhdl
count ← 0
do while count < 16
  count ← count + 1
end
GPR[RT] ← EXTZ64(count)
```

Let `count` be the number of contiguous trailing byte elements of `VSR[VRB+32]` having a zero least-significant bit.

**Special Registers Altered:**

None
6.12.5 Vector Bit Insert/Extract Instructions

Vector Parallel Bits Deposit Doubleword VX-form

### vpdepd VRT,VRA,VRB

|--------|----------------------|----------------------|----------------------|----------------------|

if MSR.VEC=0 then Vector_Unavailable()

\[
\text{do } i = 0 \text{ to } 1 \\
\text{VS}R[VRT+32].\text{dword}[i] \leftarrow 0 \\
\text{m} \leftarrow 0 \\
\text{k} \leftarrow 0 \\
\text{do while}(m < 64) \\
\quad \text{if VSR[VRB+32].dword}[i].\text{bit}[63-m]=1 \text{ then do} \\
\quad \quad \text{result} \leftarrow \text{VSR[VRA+32].dword}[i].\text{bit}[63-k] \\
\quad \quad \text{VSR[VRT+32].dword}[i].\text{bit}[63-m] \leftarrow \text{result} \\
\quad \quad \text{k} \leftarrow k + 1 \\
\quad \text{end} \\
\quad \text{m} \leftarrow m + 1 \\
\quad \text{end} \\
\text{end}
\]

For each integer value \(i\) from 0 to 1, do the following.

Let \(n\) be the number of bits in doubleword element \(i\) of VSR[VRB+32] that contain a 1.

The contents of the rightmost \(n\) bits of doubleword element \(i\) of VSR[VRB+32] are placed into doubleword element \(i\) of VSR[VRT+32] under control of the mask in doubleword element \(i\) of VSR[VRB+32] as follows.

- The contents of bit 63 of doubleword element \(i\) of VSR[VRA+32] are placed into the bit in doubleword element \(i\) of VSR[VRT+32] corresponding to the rightmost bit in doubleword element \(i\) of VSR[VRB+32] that contains a 1 (if any).

- The contents of bit 62 of doubleword element \(i\) of VSR[VRA+32] are placed into the bit in doubleword element \(i\) of VSR[VRT+32] corresponding to the second rightmost bit in doubleword element \(i\) of VSR[VRB+32] that contains a 1 (if any), and so forth until

- The contents of bit \(64-n\) of doubleword element \(i\) of VSR[VRA+32] are placed into the bit in doubleword element \(i\) of VSR[VRT+32] corresponding to the leftmost bit in doubleword element \(i\) of VSR[VRB+32] that contains a 1 (if any).

The contents of bits in doubleword element \(i\) of VSR[VRT+32] corresponding to bits in doubleword element \(i\) of VSR[VRB+32] that contain a 0 are set to 0.

### Special Registers Altered:

None
Vector Parallel Bits Extract Doubleword VX-form

vpextd  VRT,VRA,VRB

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>VRT</td>
<td>14</td>
<td>VRA</td>
<td>21</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
  result ← 0
  m ← 0
  k ← 0
  do while(m < 64)
    if VSR[VRB+32].dword[i].bit[63-m]=1 then do
      result.bit[63-k] ← VSR[VRA+32].dword[i].bit[63-m]
      k ← k + 1
    end
    m ← m + 1
  end
  VSR[VRT+32].dword[i] ← result
end

For each integer value i from 0 to 1, do the following.
Starting from the right, for each bit in doubleword element i of VSR[VRB+32] that is equal to 1, place the contents of the corresponding bit in doubleword element i of VSR[VRA+32] into the rightmost unoccupied bit of doubleword element i of VSR[VRT+32]. Any bits in doubleword element i of VSR[VRT+32] to the left of the most-significant bit copied are set to 0.

Special Registers Altered:
None

Register Data Layout for vpextd

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRA+32].dword[0]</td>
<td>VSR[VRA+32].dword[1]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>
6.12.6 Vector Centrifuge Instruction

**Vector Centrifuge Doubleword VX-form**

```
vcfuged VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>i</th>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
  ptr0 ← 0
  ptr1 ← 0
  do j = 0 to 63
    if VSR[VRB+32].dword[i].bit[j]=0b0 then do
      result.bit[ptr0] ← VSR[VRA+32].dword[i].bit[j]
      ptr0 ← ptr0 + 1
    end
    if VSR[VRB+32].dword[i].bit[63-j]=1 then do
      result.bit[63-ptr1] ← VSR[VRA+32].dword[i].bit[63-j]
      ptr1 ← ptr1 + 1
    end
  end
  VSR[VRT+32].dword[i] ← result
end

For each doubleword element i of VSR[VRA+32], the bits whose corresponding bits in the mask in doubleword element i of VSR[VRB+32] equal 1 are placed in the rightmost bits in doubleword element i of VSR[VRT+32], maintaining their original relative order. The other bits in doubleword element i of VSR[VRA+32] are placed in the leftmost bits in doubleword element i of VSR[VRT+32], maintaining their original relative order.

**Special Registers Altered:**

None

**Register Data Layout for vcfuged**

```
src1         VSR[VRA+32].dword[0]  VSR[VRA+32].dword[1]
src2         VSR[VRB+32].dword[0]  VSR[VRB+32].dword[1]
result       VSR[VRT+32].dword[0]  VSR[VRT+32].dword[1]
```
6.12.7 Vector Population Count Instructions

**Vector Population Count Byte VX-form**

```
vpopcntb VRT,VRB
```

If MSR.VEC=0 then Vector_Unavailable()

```
if MSR.VEC=0 then Vector_Unavailable()
    do i = 0 to 15
       n ← 0
       do j = 0 to 7
           n ← n + VSR[VRB+32].byte[i].bit[j]
       end
       VSR[VRT+32].byte[i] ← n
    end
```

For each integer value from 0 to 15, do the following.
A count of the number of bits set to 1 in byte element i of VSR[VRB+32] is placed into byte element i of VSR[VRT+32]. This number ranges from 0 to 8, inclusive.

**Special Registers Altered:**
None

**Register Data Layout for vpopcntb**

<table>
<thead>
<tr>
<th>src</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>.byte[0]</td>
<td>.byte[0]</td>
</tr>
<tr>
<td>.byte[8]</td>
<td>.byte[8]</td>
</tr>
<tr>
<td>.byte[12]</td>
<td>.byte[12]</td>
</tr>
<tr>
<td>.byte[14]</td>
<td>.byte[14]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td>.byte[0]</td>
<td>VSR[VRB+32].byte[0]</td>
</tr>
</tbody>
</table>

**Vector Population Count Halfword VX-form**

```
vpopcnth VRT,VRB
```

If MSR.VEC=0 then Vector_Unavailable()

```
if MSR.VEC=0 then Vector_Unavailable()
    do i = 0 to 7
       n ← 0
       do j = 0 to 15
           n ← n + VSR[VRB+32].hword[i].bit[j]
       end
       VSR[VRT+32].hword[i] ← n
    end
```

For each integer value i from 0 to 7, do the following.
A count of the number of bits set to 1 in halfword element i of VSR[VRB+32] is placed into halfword element i of VSR[VRT+32]. This number ranges from 0 to 16, inclusive.

**Special Registers Altered:**
None

**Register Data Layout for vpopcnth**

<table>
<thead>
<tr>
<th>src</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32].hword[0]</td>
<td>VSR[VRB+32].hword[0]</td>
</tr>
<tr>
<td>VSR[VRT+32].hword[0]</td>
<td>VSR[VRT+32].hword[0]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32].hword[0]</td>
<td>VSR[VRT+32].hword[0]</td>
</tr>
</tbody>
</table>
Vector Population Count Word VX-form

\[ \text{vpopcntw } \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>6</th>
<th>VRB</th>
<th>1923</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[ \text{do } i = 0 \text{ to } 3 \]
\[ n \leftarrow 0 \]
\[ \text{do } j = 0 \text{ to } 31 \]
\[ n \leftarrow n + \text{VSR}[\text{VRB+32}].\text{word}[i].\text{bit}[j] \]
end

\[ \text{VSR}[\text{VRT+32}].\text{word}[i] \leftarrow n \]
end

For each integer value \( i \) from 0 to 3, do the following.
A count of the number of bits set to 1 in word element \( i \) of \( \text{VSR}[\text{VRB+32}] \) is placed into word element \( i \) of \( \text{VSR}[\text{VRT+32}] \). This number ranges from 0 to 32, inclusive.

Special Registers Altered:
None

Register Data Layout for vpopcntw

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 32</td>
<td>64 96 127</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector Population Count Doubleword VX-form

\[ \text{vpopcntd } \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>6</th>
<th>VRB</th>
<th>1987</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[ \text{do } i = 0 \text{ to } 1 \]
\[ n \leftarrow 0 \]
\[ \text{do } j = 0 \text{ to } 63 \]
\[ n \leftarrow n + \text{VSR}[\text{VRB+32}].\text{dword}[i].\text{bit}[j] \]
end

\[ \text{VSR}[\text{VRT+32}].\text{dword}[i] \leftarrow n \]
end

For each integer value \( i \) from 0 to 1, do the following.
A count of the number of bits set to 1 in doubleword element \( i \) of \( \text{VSR}[\text{VRB+32}] \) is placed into doubleword element \( i \) of \( \text{VSR}[\text{VRT+32}] \). This number ranges from 0 to 64, inclusive.

Special Registers Altered:
None
### 6.12.8 Vector Parity Byte Instructions

**Vector Parity Byte Word VX-form**

```plaintext
vprtybw VRT,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
  s ← 0
  do j = 0 to 3
    s ← s ^ VSR[VRB+32].word[i].byte[j].bit[7]
  end
  VSR[VRT+32].word[i] ← CHOP32(EXTZ(s))
end
```

For each integer value `i` from 0 to 3, do the following:

- If the sum of the least significant bit in each byte sub-element of word element `i` of `VSR[VRB+32]` is odd, the value 1 is placed into word element `i` of `VSR[VRT+32]`; otherwise the value 0 is placed into word element `i` of `VSR[VRT+32]`.

**Special Registers Altered:**

None

**Register Data Layout for vprtybw**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32].word[0]</td>
<td>VSR[VRB+32].word[1]</td>
<td>VSR[VRT+32].word[0]</td>
</tr>
</tbody>
</table>

**Vector Parity Byte Doubleword VX-form**

```plaintext
vprtybd VRT,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
  s ← 0
  do j = 0 to 7
    s ← s ^ VSR[VRB+32].dword[i].byte[j].bit[7]
  end
  VSR[VRT+32].dword[i] ← CHOP64(EXTZ(s))
end
```

For each integer value `i` from 0 to 1, do the following:

- If the sum of the least significant bit in each byte sub-element of doubleword element `i` of `VSR[VRB+32]` is odd, the value 1 is placed into doubleword element `i` of `VSR[VRT+32]`; otherwise the value 0 is placed into doubleword element `i` of `VSR[VRT+32]`.

**Special Registers Altered:**

None

**Register Data Layout for vprtybd**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32].dword[0]</td>
<td>VSR[VRB+32].dword[1]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>
Power ISA™ I

Vector Parity Byte Quadword VX-form

\( \text{vprtybq} \quad \text{VRT,VRB} \)

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th></th>
<th>10</th>
<th></th>
<th>15</th>
<th></th>
<th>21</th>
<th></th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>VRT</td>
<td></td>
<td>10</td>
<td>VRB</td>
<td>15</td>
<td>VRT</td>
<td>31</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\( s \leftarrow 0 \)

\( \text{do } j = 0 \text{ to } 15 \)

\( s \leftarrow s \oplus \text{VSR[VRB+32].byte[j].bit[7]} \)

\( \text{end} \)

\( \text{VSR[VRT+32]} \leftarrow \text{CHOP128(EXTZ(s))} \)

If the sum of the least significant bit in each byte element of \( \text{VSR[VRB+32]} \) is odd, the value 1 is placed into \( \text{VSR[VRT+32]} \); otherwise the value 0 is placed into \( \text{VSR[VRT+32]} \).

Special Registers Altered:

None

Register Data Layout for vprtybq

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

0  64  127
6.12.9 Vector Bit Permute Instructions

Vector Bit Permute Doubleword VX-form

vbpermd VRT,VRA,VRB

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1484</th>
<th>31</th>
</tr>
</thead>
</table>

If MSR.VEC=0 then Vector_Unavailable()

For each integer value i from 0 to 1, and for each integer value j from 0 to 7, do the following.

Let index be the contents of byte sub-element j of doubleword element i of VSR[VRB+32].

If index is less than 64, then the contents of bit index of doubleword i of VSR[VRA+32] are placed into bit 56+j of doubleword element i of VSR[VRT+32]. Otherwise, bit 56+j of doubleword element i of VSR[VRT+32] is set to 0.

The contents of bits 0:55 of doubleword element i of VSR[VRT+32] are set to 0.

Special Registers Altered:

None

Register Data Layout for vbpermd

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRA+32].dword[0]</td>
<td>VSR[VRA+32].dword[0]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>
Vector Bit Permute Quadword VX-form

vbpermq VRT, VRA, VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1356</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

for i from 0 to 15
index ← VSR[VRB+32].byte[i]
if index < 128 then
perm.bit[i] ← VSR[VRA+32].bit[index]
else
perm.bit[i] ← 0
end
VSR[VRT+32].dword[0] ← CHOP64(EXTZ(perm))
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000

For each integer value i from 0 to 15, do the following.

Let index be the contents of byte element i of VSR[VRB+32].

If index is less than 128, then the contents of bit index of VSR[VRB+32] are placed into bit 48+i of doubleword element i of VSR[VRT+32]. Otherwise, bit 48+i of doubleword element i of VSR[VRT+32] is set to 0.

The contents of bits 0:47 of VSR[VRT+32] are set to 0.
The contents of bits 64:127 of VSR[VRT+32] are set to 0.

Special Registers Altered:
None

Register Data Layout for mtvsrqm

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>

Programming Note

The fact that the permuted bit is 0 if the corresponding index value exceeds 127 permits the permuted bits to be selected from a 256-bit quantity, using a single index register. For example, assume that the 256-bit quantity Q, from which the permuted bits are to be selected, is in registers v2 (high-order 128 bits of Q) and v3 (low-order 128 bits of Q), that the index values are in register v1, with each byte of v1 containing a value in the range 0:255, and that each byte of register v4 contains the value 128. The following code sequence selects eight permuted bits from Q and places them into the low-order byte of v6.

vbpermq v6, v1, v2  # select from high-order half of Q
vxor v0, v1, v4    # adjust index values
vbpermq v5, v0, v3  # select from low-order half of Q
vor v6, v6, v5     # merge the two selections
### 6.13 Vector Mask Manipulation Instructions

#### 6.13.1 Vector Mask Move Instructions

The Vector Mask Move instructions support creating a field mask in a VSR from a bit mask specified either in a GPR or as an immediate operand.

#### Move to VSR Byte Mask VX-form

**mtvsrbm VRT,RB**

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>16</th>
<th>RB</th>
<th>1602</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
  if GPR[RB].bit[48+i]=0 then
    VSR[VRT+32].byte[i] ← 0x00
  else
    VSR[VRT+32].byte[i] ← 0xFF
  end

Let bm be the contents of bits 48:63 of GPR[RB].

For each integer value i from 0 to 15, do the following.

The contents of byte element i of VSR[VRT+32] is set to all 0s if bit i of bm is equal to 0.

The contents of byte element i of VSR[VRT+32] is set to all 1s if bit i of bm is equal to 1.

Special Registers Altered:

None

#### Move to VSR Halfword Mask VX-form

**mtvsrhm VRT,RB**

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>17</th>
<th>RB</th>
<th>1602</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
  if GPR[RB].bit[56+i]=0 then
    VSR[VRT+32].hword[i] ← 0x0000
  else
    VSR[VRT+32].hword[i] ← 0xFFFF
  end

Let bm be the contents of bits 56:63 of GPR[RB].

For each integer value i from 0 to 7, do the following.

The contents of halfword element i of VSR[VRT+32] is set to all 0s if bit i of bm is equal to 0.

The contents of halfword element i of VSR[VRT+32] is set to all 1s if bit i of bm is equal to 1.

Special Registers Altered:

None

#### Register Data Layout for vcntmbb

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
</tr>
</tbody>
</table>

#### Register Data Layout for vcntmbh

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
<td>[6]</td>
<td>[7]</td>
</tr>
</tbody>
</table>

Chapter 6. Vector Facility 463
Move to VSR Word Mask VX-form

\texttt{mtvsrwm} \texttt{VRT,RB}

<table>
<thead>
<tr>
<th>\texttt{0}</th>
<th>\texttt{6}</th>
<th>\texttt{11}</th>
<th>\texttt{16}</th>
<th>\texttt{21}</th>
<th>\texttt{1602}</th>
</tr>
</thead>
</table>

\texttt{if MSR.VEC=0 then VectorUnavailable()}

\texttt{do i = 0 to 3}
\texttt{if GPR[RB].bit[60+i]=0 then}
\texttt{VSR[VRT+32].word[i] \leftarrow 0x0000_0000}
\texttt{else}
\texttt{VSR[VRT+32].word[i] \leftarrow 0xFFFF_FFFF}
\texttt{end}

Let \texttt{bm} be the contents of bits 60:63 of \texttt{GPR[RB]}

For each integer value \texttt{i} from 0 to 3, do the following.
The contents of word element \texttt{i} of \texttt{VSR[VRT+32]} is set to all 0s if bit \texttt{i} of \texttt{bm} is equal to 0.
The contents of word element \texttt{i} of \texttt{VSR[VRT+32]} is set to all 1s if bit \texttt{i} of \texttt{bm} is equal to 1.

Special Registers Altered:
None

Move to VSR Doubleword Mask VX-form

\texttt{mtvsrdm} \texttt{VRT,RB}

<table>
<thead>
<tr>
<th>\texttt{0}</th>
<th>\texttt{6}</th>
<th>\texttt{11}</th>
<th>\texttt{16}</th>
<th>\texttt{21}</th>
<th>\texttt{1602}</th>
<th>\texttt{31}</th>
</tr>
</thead>
</table>

\texttt{if MSR.VEC=0 then VectorUnavailable()}

\texttt{do i = 0 to 1}
\texttt{if GPR[RB].bit[62+i]=0 then}
\texttt{VSR[VRT+32].dword[i] \leftarrow 0x0000_0000_0000_0000}
\texttt{else}
\texttt{VSR[VRT+32].dword[i] \leftarrow 0xFFFF_FFFF_FFFF_FFFF}
\texttt{end}

Let \texttt{bm} be the contents of bits 62:63 of \texttt{GPR[RB]}

For each integer value \texttt{i} from 0 to 1, do the following.
The contents of doubleword element \texttt{i} of \texttt{VSR[VRT+32]} is set to all 0s if bit \texttt{i} of \texttt{bm} is equal to 0.
The contents of doubleword element \texttt{i} of \texttt{VSR[VRT+32]} is set to all 1s if bit \texttt{i} of \texttt{bm} is equal to 1.

Special Registers Altered:
None

Register Data Layout for \texttt{mtvsrwm}

<table>
<thead>
<tr>
<th>src</th>
<th>GPR[RB]</th>
</tr>
</thead>
</table>

\texttt{result}

\begin{array}{cccc}
\texttt{VSR[VRT+32].word[0]} & \texttt{VSR[VRT+32].word[1]} & \texttt{VSR[VRT+32].word[2]} & \texttt{VSR[VRT+32].word[3]}
\end{array}

| 0 | 32 | 64 | 96 | 127 |

Register Data Layout for \texttt{mtvsrdm}

<table>
<thead>
<tr>
<th>src</th>
<th>GPR[RB]</th>
</tr>
</thead>
</table>

\texttt{result}

\begin{array}{c}
\texttt{VSR[VRT+32].dword[0]}
\end{array}

\begin{array}{c}
\texttt{VSR[VRT+32].dword[1]}
\end{array}

| 0 | 64 | 127 |
Move to VSR Quadword Mask VX-form

mtvsrqm VRT, RB

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>10</th>
<th>16</th>
<th>20</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

if GPR[R].bit[63]=0 then
    VSR[VRT+32] ← 0x0000_0000_0000_0000_0000_0000_0000_0000
else
    VSR[VRT+32] ← 0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF

Let bm be the contents of bits 63 of GPR[R].

The contents of VSR[VRT+32] is set to all 0s if bm is equal to 0.

The contents of VSR[VRT+32] is set to all 1s if bm is equal to 1.

Special Registers Altered:
None

Move To VSR Byte Mask Immediate DX-form

mtvsrbmi VRT, b0

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

bm.bit[0:9] ← b0
bm.bit[10:14] ← b1
bm.bit[15] ← b2
do i = 0 to 15
    if bm.bit[i]=0 then
        VSR[VRT+32].byte[i] ← 0x00
    else
        VSR[VRT+32].byte[i] ← 0xFF
end

Let bm be the concatenation of b0, b1 and b2.

For each integer value i from 0 to 15, do the following.

The contents of byte element i of VSR[VRT+32] is set to all 0s if bit i of bm is equal to 0.

The contents of byte element i of VSR[VRT+32] is set to all 1s if bit i of bm is equal to 1.

Special Registers Altered:
None

Register Data Layout for mtvsrqm

src GPR[R]
result VSR[VRT+32]

Register Data Layout for mtvsrbmi

6.13.2 Vector Expand Mask Instructions

The Vector Expand Mask instructions support creating a field mask by replicating the contents of bit 0 of each element in the source VSR to all bits in the corresponding element in the target VSR.

**Vector Expand Byte Mask VX-form**

\[
\text{vexpandbm } \text{VRT}, \text{VRB}
\]

\[
\begin{array}{cccccccc}
\text{VRT} & 0 & 11 & 0 & 16 & 21 & 1602 & 31 \\
\text{VRB} & 4 & 6 & 16 & 16 & 16 & 16 & 16
\end{array}
\]

- if MSR.VEC=0 then Vector_Unavailable()
- do i = 0 to 15
  - if VSR[VRB+32].byte[i].bit[0]=1 then
    - VSR[VRT+32].byte[i] ← 0xFF
  - else
    - VSR[VRT+32].byte[i] ← 0x00
- end

For each integer value \(i\) from 0 to 15, do the following.

Let \(bmi\) be the contents of bit 0 of byte element \(i\) of VSR[VRB+32].

The contents of byte element \(i\) of VSR[VRT+32] are set to all 0s if \(bmi\) is equal to 0.

The contents of byte element \(i\) of VSR[VRT+32] are set to all 1s if \(bmi\) is equal to 1.

**Special Registers Altered:**

None

**Register Data Layout for vexpandbm**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
</tr>
</tbody>
</table>

**Vector Expand Halfword Mask VX-form**

\[
\text{vexpandhm } \text{VRT}, \text{VRB}
\]

\[
\begin{array}{cccccccc}
\text{VRTX} & 0 & 11 & 0 & 16 & 21 & 1602 & 31 \\
\text{VRB} & 4 & 6 & 16 & 16 & 16 & 16 & 16
\end{array}
\]

- if MSR.VEC=0 then Vector_Unavailable()
- do i = 0 to 7
  - if VSR[VRB+32].hword[i].bit[0]=1 then
    - VSR[VRT+32].hword[i] ← 0xFFFF
  - else
    - VSR[VRT+32].hword[i] ← 0x0000
- end

For each integer value \(i\) from 0 to 7, do the following.

Let \(bmi\) be the contents of bit 0 of halfword element \(i\) of VSR[VRB+32].

The contents of halfword element \(i\) of VSR[VRT+32] are set to all 0s if \(bmi\) is equal to 0.

The contents of halfword element \(i\) of VSR[VRT+32] are set to all 1s if \(bmi\) is equal to 1.

**Special Registers Altered:**

None

**Register Data Layout for vexpandhm**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
</tr>
</tbody>
</table>
Vector Expand Word Mask VX-form
vexpandwm VRT,VRB

if MSR.VEC=0 then Vector_Unavailable()
do i = 0 to 3
  if VSR[VRB+32].word[i].bit[0]=1 then
    VSR[VRT+32].word[i] ← 0xFFFF_FFFF
  else
    VSR[VRT+32].word[i] ← 0x0000_0000
  end

For each integer value i from 0 to 3, do the following.
Let bmi be the contents of bit 0 of word element i of VSR[VRB+32].
The contents of word element i of VSR[VRT+32] are set to all 0s if bmi is equal to 0.
The contents of word element i of VSR[VRT+32] are set to all 1s if bmi is equal to 1.

Special Registers Altered:
None

Register Data Layout for vcntmbw

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
</tr>
</tbody>
</table>

Vector Expand Doubleword Mask VX-form
vexpanddm VRT,VRB

if MSR.VEC=0 then Vector_Unavailable()
do i = 0 to 1
  if VSR[VRB+32].dword[i].bit[0]=1 then
    VSR[VRT+32].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
  else
    VSR[VRT+32].dword[i] ← 0x0000_0000_0000_0000
  end

For each integer value i from 0 to 1, do the following.
Let bmi be the contents of bit 0 of doubleword element i of VSR[VRB+32],
The contents of doubleword element i of VSR[VRT+32] are set to all 0s if bmi is equal to 0.
The contents of doubleword element i of VSR[VRT+32] are set to all 1s if bmi is equal to 1.

Special Registers Altered:
None

Register Data Layout for vcntmbd

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32].dword[0]</th>
<th>VSR[VRB+32].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>VSR[VRT+32].dword[0]</td>
<td>VSR[VRT+32].dword[1]</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>
Vector Expand Quadword Mask VX-form

\[ \text{vexpandqm} \rightarrow \text{VRT}, \text{VRB} \]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>4</th>
<th>VRB</th>
<th>16</th>
<th>21</th>
<th>1602</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then VectorUnavailable()

if VSR[VRB+32].bit[0]=1 then

\[ \text{VSR}[VRT+32] \leftarrow 0xFFFFFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF \]

else

\[ \text{VSR}[VRT+32] \leftarrow 0x0000_0000_0000_0000_0000_0000_0000_0000 \]

Let \( \text{bmi} \) be the contents of bit 0 of VSR[VRB+32].

The contents of VSR[VRT+32] are set to all 0s if bmi is equal to 0.

The contents of VSR[VRT+32] are set to all 1s if bmi is equal to 1.

Special Registers Altered:

None

Register Data Layout for vexpandqm

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>127</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>127</td>
</tr>
</tbody>
</table>
6.13.3 Vector Count Mask Bits Instructions

The Vector Count Mask Bits instructions count the number of true (or false) mask bits (bit 0 of each element) in a VSR and place the count in the leftmost byte of a GPR (i.e., can be used by Load VSX Vector with Length and Store Vector with Length).

**Vector Count Mask Bits Byte VX-form**

`vcntmbb RT,VRB,MP`

```
if MSR.VEC=0 then Vector_Unavailable()

count = 0
do i = 0 to 15
    count ← count + EXTZ8(VSR[VRB+32].byte[i].bit[0]=MP)
end

GPR[RT] ← count << 56
```

The number of byte elements having bit 0 set to the value `MP` in VSR[VRB+32] is placed into bits 0:7 of GPR[RT]. Bits 8:63 of GPR[RT] are set to 0.

**Special Registers Altered:**

None

**Register Data Layout for vcntmbb**

```
result | GPR[RT]
0   | 8  16  32  48  56  64  72  80  88  96 104 112 120 127
```

**Vector Count Mask Bits Halfword VX-form**

`vcntmbh RT,VRB,MP`

```
if MSR.VEC=0 then Vector_Unavailable()

count = 0
do i = 0 to 7
    count ← count + EXTZ64(VSR[VRB+32].hword[i].bit[0]=MP)
end

GPR[RT] ← count << 57
```

The number of halfword elements having bit 0 set to the value `MP` in VSR[VRB+32] is placed into bits 0:6 of GPR[RT]. Bits 7:63 of GPR[RT] are set to 0.

**Special Registers Altered:**

None

**Register Data Layout for vcntmbh**

```
result | GPR[RT]
0   | 16  32  48  56  64  72  80  88  96 104 112 120 127
```
**Vector Count Mask Bits Word VX-form**

\[
\begin{array}{cccccc}
\text{vcntmbw} & \text{RT} & \text{VRB} & \text{MP} & \text{VRB}+32 & \text{VRB}+32 \\
0 & 6 & 11 & 16 & 21 & 1602 \\
\end{array}
\]

\[
\text{if MSR.VEC=0 then Vector.Unavailable()}
\]

\[
\text{count} = 0
\]

\[
\text{do i = 0 to 3}
\]

\[
\text{count} \leftarrow \text{count} + \text{EXTZ64(VSR[VRB+32].word[i].bit[0]=MP)}
\]

\[
\text{end}
\]

\[
\text{GPR}[\text{RT}] \leftarrow \text{count} \ll 58
\]

The number of word elements having bit 0 set to the value \text{MP} in VSR[\text{VRB+32}] is placed into bits 0:5 of GPR[\text{RT}]. Bits 6:63 of GPR[\text{RT}] are set to 0.

**Special Registers Altered:**

None


**Register Data Layout for \text{vcntmbw}**

\[
\begin{array}{lllll}
\text{src} & \text{VSR[VRB+32].word[0]} & \text{VSR[VRB+32].word[1]} & \text{VSR[VRB+32].word[2]} & \text{VSR[VRB+32].word[3]} \\
\text{result} & \text{GPR[RT]} & \text{127} & \text{96} & \text{64} & \text{32} & \text{0} \\
\end{array}
\]


**Vector Count Mask Bits Doubleword VX-form**

\[
\begin{array}{cccccc}
\text{vcntmbd} & \text{RT} & \text{VRB} & \text{MP} & \text{VRB}+32 & \text{VRB}+32 \\
0 & 6 & 11 & 16 & 21 & 1602 \\
\end{array}
\]

\[
\text{if MSR.VEC=0 then Vector.Unavailable()}
\]

\[
\text{count} = 0
\]

\[
\text{do i = 0 to 1}
\]

\[
\text{count} \leftarrow \text{count} + \text{EXTZ64(VSR[VRB+32].dword[i].bit[0]=MP)}
\]

\[
\text{end}
\]

\[
\text{GPR}[\text{RT}] \leftarrow \text{count} \ll 59
\]

The number of doubleword elements having bit 0 set to the value \text{MP} in VSR[\text{VRB+32}] is placed into bits 0:4 of GPR[\text{RT}]. Bits 5:63 of GPR[\text{RT}] are set to 0.

**Special Registers Altered:**

None


**Register Data Layout for \text{vcntmbd}**

\[
\begin{array}{llll}
\text{src} & \text{VSR[VRB+32].dword[0]} & \text{VSR[VRB+32].dword[1]} \\
\text{result} & \text{GPR[RT]} & \text{127} & \text{64} \\
\end{array}
\]
### 6.13.4 Vector Extract Mask Instructions

The Vector Extract Mask instructions extracts bit 0 of each element from a VSR into a GPR.

#### Vector Extract Byte Mask VX-form

**vextractbm**

\[
\text{RT}, \text{VRB}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>16</th>
<th>21</th>
<th>1602</th>
</tr>
</thead>
</table>

**Vector Extract Byte Mask VX-form**

\[
\text{RT}, \text{VRB}
\]

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>21</th>
<th>1602</th>
</tr>
</thead>
</table>

**Register Data Layout for vextractbm**

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
\text{result} & \text{GPR}[\text{RT}]
\end{array}
\]

**Register Data Layout for vextracthm**

\[
\begin{array}{cccccccccccccccccccc}
\text{src} & \text{VSR}[\text{VRB}+32].\text{hword}[0] & \text{VSR}[\text{VRB}+32].\text{hword}[1] & \text{VSR}[\text{VRB}+32].\text{hword}[2] & \text{VSR}[\text{VRB}+32].\text{hword}[3] & \text{VSR}[\text{VRB}+32].\text{hword}[4] & \text{VSR}[\text{VRB}+32].\text{hword}[5] & \text{VSR}[\text{VRB}+32].\text{hword}[6] & \text{VSR}[\text{VRB}+32].\text{hword}[7]
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
\text{result} & \text{GPR}[\text{RT}]
\end{array}
\]

The contents of bit 0 of each byte element of VSR[VRB+32] are concatenated and placed into bits 48:63 of GPR[RT]. Bits 0:47 of GPR[RT] are set to 0.

**Special Registers Altered:**

None

#### Vector Extract Halfword Mask VX-form

**vextracthm**

\[
\text{RT}, \text{VRB}
\]

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>21</th>
<th>1602</th>
</tr>
</thead>
</table>

**Vector Extract Halfword Mask VX-form**

\[
\text{RT}, \text{VRB}
\]

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>21</th>
<th>1602</th>
</tr>
</thead>
</table>

**Register Data Layout for vextracthm**

\[
\begin{array}{cccccccccccccccccccc}
\text{src} & \text{VSR}[\text{VRB}+32].\text{hword}[0] & \text{VSR}[\text{VRB}+32].\text{hword}[1] & \text{VSR}[\text{VRB}+32].\text{hword}[2] & \text{VSR}[\text{VRB}+32].\text{hword}[3] & \text{VSR}[\text{VRB}+32].\text{hword}[4] & \text{VSR}[\text{VRB}+32].\text{hword}[5] & \text{VSR}[\text{VRB}+32].\text{hword}[6] & \text{VSR}[\text{VRB}+32].\text{hword}[7]
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
\text{result} & \text{GPR}[\text{RT}]
\end{array}
\]

The contents of bit 0 of each halfword element of VSR[VRB+32] are concatenated and placed into bits 56:63 of GPR[RT]. Bits 0:55 of GPR[RT] are set to 0.

**Special Registers Altered:**

None
Vector Extract Word Mask VX-form

\textit{vextractwm} \quad \text{RT,VRB}

\begin{align*}
\text{if MSR.VEC=0 then Vector\_Unavailable}()\\
\text{do } i = 0 \text{ to } 3 \\
\quad \GPR[RT].\text{bit}[60+i] &\leftarrow \VSR[VRB+32].\text{word}[i].\text{bit}[0]\\
\text{end}\\
\GPR[RT].\text{bit}[0:59] &\leftarrow 0
\end{align*}

The contents of bit 0 of each word element of \( \VSR[VRB+32] \) are concatenated and placed into bits 60:63 of \( \GPR[RT] \). Bits 0:59 of \( \GPR[RT] \) are set to 0.

\textbf{Special Registers Altered:} \\
None

\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{VSR[VRB+32].word[0]} & \textbf{VSR[VRB+32].word[1]} & \textbf{VSR[VRB+32].word[2]} & \textbf{VSR[VRB+32].word[3]} \\
\hline
0 & 32 & 64 & 96 & 127 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
\textbf{src} & \textbf{result} \\
\hline
\textbf{VSR[VRB+32].word[0]} & \textbf{GPR[RT]} \\
\hline
0 & 64 & 127 \\
\hline
\end{tabular}

Vector Extract Doubleword Mask VX-form

\textit{vextractdm} \quad \text{RT,VRB}

\begin{align*}
\text{if MSR.VEC=0 then Vector\_Unavailable}()\\
\text{do } i = 0 \text{ to } 1 \\
\quad \GPR[RT].\text{bit}[62+i] &\leftarrow \VSR[VRB+32].\text{dword}[i].\text{bit}[0]\\
\text{end}\\
\GPR[RT].\text{bit}[0:61] &\leftarrow 0
\end{align*}

The contents of bit 0 of each doubleword element of \( \VSR[VRB+32] \) are concatenated and placed into bits 62:63 of \( \GPR[RT] \). Bits 0:61 of \( \GPR[RT] \) are set to 0.

\textbf{Special Registers Altered:} \\
None

\begin{tabular}{|c|c|c|c|}
\hline
\textbf{VSR[VRB+32].dword[0]} & \textbf{VSR[VRB+32].dword[1]} \\
\hline
0 & 64 & 127 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
\textbf{src} & \textbf{result} \\
\hline
\textbf{VSR[VRB+32].dword[0]} & \textbf{GPR[RT]} \\
\hline
0 & 64 & 127 \\
\hline
\end{tabular}
Vector Extract Quadword Mask VX-form

vextractqm RT,VRB

<table>
<thead>
<tr>
<th>src</th>
<th>VS[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>GPR[RT]</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

GPR[RT] ← EXTZ64(VSR[VRB+32].bit[0])

The contents of bit 0 of VSR[VRB+32] are placed into bit 63 of GPR[RT]. Bits 0:62 of GPR[RT] are set to 0.

Special Registers Altered:
None
6.14 Vector String Instructions

6.14.1 Vector String Isolate Instructions

**Vector String Isolate Byte Right-justified VX-form**

\[
\text{vstribr} \quad \text{VRT}, \text{VRB} \quad (Rc=0)
\]

\[
\text{vstribr.} \quad \text{VRT}, \text{VRB} \quad (Rc=1)
\]

\[
\begin{array}{cccccccc}
\text{byte} & \text{byte} & \text{byte} & \text{byte} & \text{byte} & \text{byte} & \text{byte} & \text{byte} \\
0 & 6 & 11 & 16 & 21 & 22 & 31 & \\
\end{array}
\]

if MSR.VEC=0 then VectorUnavailable();

null_found ← 0

while(!null_found) do i = 0 to 15

null_found ← (VSR[VRB+32].byte[15-i]=0)

end

do j = i to 15

VSR[VRT+32].byte[15-j] ← 0
end

if Rc=1 then

CR.field[6] ← 0b00 || null_found || 0b0

From right-to-left, the contents of each byte element of VSR[VRB+32] are placed into the corresponding byte element in VSR[VRT+32]. If a byte element in VSR[VRB+32] is found to contain 0, the corresponding byte element and all byte elements to the left of that byte element in VSR[VRT+32] are set to 0.

**Special Registers Altered:**

CR field 6 (if Rc=1)

**Register Data Layout for vstribr[.] & vstribl[.]**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
<td>127</td>
</tr>
</tbody>
</table>

**Vector String Isolate Byte Left-justified VX-form**

\[
\text{vstribl} \quad \text{VRT}, \text{VRB} \quad (Rc=0)
\]

\[
\text{vstribl.} \quad \text{VRT}, \text{VRB} \quad (Rc=1)
\]

\[
\begin{array}{cccccccc}
\text{byte} & \text{byte} & \text{byte} & \text{byte} & \text{byte} & \text{byte} & \text{byte} & \text{byte} \\
0 & 6 & 11 & 16 & 21 & 22 & 31 & \\
\end{array}
\]

if MSR.VEC=0 then VectorUnavailable();

null_found ← 0

while(!null_found) do i = 0 to 15

null_found ← (VSR[VRB+32].byte[i]=0)

VSR[VRT+32].byte[i] ← VSR[VRB+32].byte[i]
end

do j = i to 15

VSR[VRT+32].byte[j] ← 0
end

if Rc=1 then

CR.field[6] ← 0b00 || null_found || 0b0

From left-to-right, the contents of each byte element of VSR[VRB+32] are placed into the corresponding byte element in VSR[VRT+32]. If a byte element in VSR[VRB+32] is found to contain 0, the corresponding byte element and all byte elements to the right of that byte element in VSR[VRT+32] are set to 0.

**Special Registers Altered:**

CR field 6 (if Rc=1)
Vector String Isolate Halfword Right-justified VX-form

```
vstrihr VRT, VRB (Rc=0)
vstrihr VRT, VRB (Rc=1)
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>3</th>
<th>VRB</th>
<th>Rc</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>2</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

null_found ← 0

while(!null_found) do i = 0 to 7
null_found ← (VSR[VRB+32].hword[7-i] = 0)
VSR[VRT+32].hword[7-i] ← VSR[VRB+32].hword[7-i]
end

do j = i to 7
VSR[VRT+32].hword[7-j] ← 0
end

if Rc=1 then
CR.field[6] ← 0b00 || null_found || 0b0

From right-to-left, the contents of each halfword element of VSR[VRB+32] are placed into the corresponding halfword element in VSR[VRT+32]. If a halfword element in VSR[VRB+32] is found to contain 0, the corresponding halfword element and all halfword elements to the left of that halfword element in VSR[VRT+32] are set to 0.

Special Registers Altered:
CR field 6 (if Rc=1)

Vector String Isolate Halfword Left-justified VX-form

```
vstrihl VRT, VRB (Rc=0)
vstrihl VRT, VRB (Rc=1)
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>2</th>
<th>VRB</th>
<th>Rc</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>2</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

null_found ← 0

while(!null_found) do i = 0 to 7
null_found ← (VSR[VRB+32].hword[i] = 0)
VSR[VRT+32].hword[i] ← VSR[VRB+32].hword[i]
end

do j = i to 7
VSR[VRT+32].hword[j] ← 0
end

if Rc=1 then
CR.field[6] ← 0b00 || null_found || 0b0

From left-to-right, the contents of each halfword element of VSR[VRB+32] are placed into the corresponding halfword element in VSR[VRT+32]. If a halfword element in VSR[VRB+32] is found to contain 0, the corresponding halfword element and all halfword elements to the right of that halfword element in VSR[VRT+32] are set to 0.

Special Registers Altered:
CR field 6 (if Rc=1)

Register Data Layout for vstrihr[,] & vstrihl[,]
6.14.2 Vector Clear Bytes Instructions

**Vector Clear Leftmost Bytes VX-form**

vclrlb VRT,VRA,RB

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>15</th>
<th>RB</th>
<th>31</th>
<th>397</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

N ← (GPR[R0] > 15) ? 16: GPR[R0]

do i = 0 to N-1
end

do i = N to 15
    VSR[VRT+32].byte[15-i] ← 0x00
end

Let N be the integer value in GPR[R0], or the integer value 16 if the integer value in GPR[R0] is greater than 15.

The contents of VSR[VRA+32] are placed into VSR[VRT+32] with the leftmost 16-N bytes of VSR[VRT+32] set to 0.

**Special Registers Altered**

None

**Register Data Layout for vclrlb & vclrrb**

src1 VSR[VRA+32]

src2 GPR[R0]

result VSR[VRT+32]

0 64 127

**Vector Clear Rightmost Bytes VX-form**

vclrrb VRT,VRA,RB

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>15</th>
<th>RB</th>
<th>21</th>
<th>461</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

N ← (GPR[R0] > 15) ? 16: GPR[R0]

do i = 0 to N-1
    VSR[VRT+32].byte[i] ← VSR[VRA+32].byte[i]
end

do i = N to 15
    VSR[VRT+32].byte[i] ← 0x00
end

Let N be the integer value in GPR[R0], or the integer value 16 if the integer value in GPR[R0] is greater than 15.

The contents of VSR[VRA+32] are placed into VSR[VRT+32] with the rightmost 16-N bytes of VSR[VRT+32] set to 0.

**Special Registers Altered**

None
6.15 Decimal Integer Instructions

A valid encoding of a packed decimal integer value requires the following properties.
- Each of the 31 4-bit digits of the operand’s magnitude (bits 0:123) must be in the range 0-9.
- The sign code (bits 124:127) must be in the range 10-15.

Source operands with sign codes of 0b1010, 0b1100, 0b1110, and 0b1111 are interpreted as positive values.

Source operands with sign codes of 0b1011 and 0b1101 are interpreted as negative values.

Positive and zero results are encoded with a either sign code of 0b1100 or 0b1111, depending on the preferred sign (indicated as an immediate operand).

Negative results are encoded with a sign code of 0b1101.

6.15.1 Decimal Integer Arithmetic Instructions

The Decimal Integer Arithmetic instructions operate on decimal integer values only in signed packed decimal format. Signed packed decimal format consists of 31 4-bit base-10 digits of magnitude and a trailing 4-bit sign code. Operations are performed as sign-magnitude, and produce a decimal result placed in a VSR (i.e., \texttt{bcdadd}, \texttt{bcdsub}).
### Decimal Add Modulo VX-form

**bcdadd.** VRT, VRA, VRB, PS

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>PS</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>1</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>31</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- VSR[VRT+32] ← bcd_ADD(VSR[VRA+32], VSR[VRB+32], PS)
- CR.bit[56] ← inv_flag ? 0b0 : lt_flag
- CR.bit[57] ← inv_flag ? 0b0 : gt_flag
- CR.bit[58] ← eq_flag? 0b0 : eq_flag
- CR.bit[59] ← ox_flag | inv_flag

Let src1 be the decimal integer value in VSR[VRA+32].

Let src2 be the decimal integer value in VSR[VRB+32].

src1 is added to src2.

If the unbounded result is equal to zero, do the following.
- If PS=0, the sign code of the result is set to 0b1100.
- If PS=1, the sign code of the result is set to 0b1111.

CR field 6 is set to 0b0010.

If the unbounded result is greater than zero, do the following.
- If PS=0, the sign code of the result is set to 0b1100.
- If PS=1, the sign code of the result is set to 0b1111.

If the operation overflows, CR field 6 is set to 0b0101. Otherwise, CR field 6 is set to 0b0100.

If the unbounded result is less than zero, do the following.
- The sign code of the result is set to 0b1101.
- If the operation overflows, CR field 6 is set to 0b1000. Otherwise, CR field 6 is set to 0b1000.

The low-order 31 digits of the magnitude of the result are placed in bits 0:123 of VSR[VRT+32].

The sign code is placed in bits 124:127 of VSR[VRT+32].

If either src1 or src2 is an invalid encoding of a 31-digit signed decimal value, the result is undefined and CR field 6 is set to 0b0001.

**Special Registers Altered:**
- CR field 6

---

### Decimal Subtract Modulo VX-form

**bcdsub.** VRT, VRA, VRB, PS

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>PS</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>1</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>65</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- VSR[VRT+32] ← bcd_SUBTRACT(VSR[VRA+32], VSR[VRB+32], PS)
- CR.bit[56] ← inv_flag ? 0b0 : lt_flag
- CR.bit[57] ← inv_flag ? 0b0 : gt_flag
- CR.bit[58] ← eq_flag? 0b0 : eq_flag
- CR.bit[59] ← ox_flag | inv_flag

Let src1 be the decimal integer value in VSR[VRA+32].

Let src2 be the decimal integer value in VSR[VRB+32].

src1 is subtracted by src2.

If the unbounded result is equal to zero, do the following.
- If PS=0, the sign code of the result is set to 0b1100.
- If PS=1, the sign code of the result is set to 0b1111.

CR field 6 is set to 0b0010.

If the unbounded result is greater than zero, do the following.
- If PS=0, the sign code of the result is set to 0b1100.
- If PS=1, the sign code of the result is set to 0b1111.

If the operation overflows, CR field 6 is set to 0b0101. Otherwise, CR field 6 is set to 0b0100.

If the unbounded result is less than zero, do the following.
- The sign code of the result is set to 0b1101.
- If the operation overflows, CR field 6 is set to 0b1000. Otherwise, CR field 6 is set to 0b1000.

The low-order 31 digits of the magnitude of the result are placed in bits 0:123 of VSR[VRT+32].

The sign code is placed in bits 124:127 of VSR[VRT+32].

If either src1 or src2 is an invalid encoding of a 31-digit signed decimal value, the result is undefined and CR field 6 is set to 0b0001.

**Special Registers Altered:**
- CR field 6
Register Data Layout for bcdadd & bcdsub.

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

Software should take care when interoperability with the Decimal Floating-Point facilities is required. The register format defined for 31-digit signed decimal values employed by \texttt{bcdadd} and \texttt{bcdsub} is a single 128-bit VSR. The register format defined for 31-digit signed decimal values employed by the Decimal Floating-Point instructions \texttt{ddedpdq} and \texttt{denbcdq} is a pair of 64-bit FPRs. \texttt{xxpermdi} can be used to convert between the two register formats as well as move data between the FPR and VSR halves of the Vector-Scalar Registers.

\texttt{gew} and \texttt{fmrgow} are provided to support direct move operations in 32-bit mode.

Programming Note

\texttt{bcdsub. vTmp,vA,vB,0} can be used to compare decimal operands \texttt{vA} and \texttt{vB}. Bits 0-2 of CR field 6 will be set to indicate \texttt{vA} is less than \texttt{vB} (LT), \texttt{vA} is greater than \texttt{vB} (GT), and \texttt{vA} is equal to \texttt{vB} (EQ).

\texttt{bcdsub. vTmp,vA,vA,0} can be used to test if an operand \texttt{vA} is an invalid encoding of a decimal value.

Programming Note

When bit 3 of CR field 6 is set to 1 by \texttt{bcdadd} or \texttt{bcdsub}, either an overflow occurred or one or both operands are not valid encodings of decimal values. Discerning whether an overflow occurred can be accomplished by performing the other decimal instruction on the operands. For example, if \texttt{bcdadd} caused bit 3 of CR field 6 to be set to 1, performing \texttt{bcdsub} on the same set of operands will cause bit 3 of CR field 6 to be set to 1 if and only if one or both of the operands is an invalid encoding. If bit 3 of CR field 6 is not set by \texttt{bcdsub}, then the \texttt{bcdadd} can be asserted to have overflowed. Likewise, \texttt{bcdadd} can be used in a similar manner to determine the cause of bit 3 of CR field 6 getting set by a \texttt{bcdsub}.
6.15.2 Decimal Integer Format Conversion Instructions

Decimal Convert From National VX-form

Let src be the national decimal value in VSR[VRB+32].

src is placed in VSR[VRT+32] in packed decimal format.

A valid encoding of a national decimal value requires the following.

- The contents of halfword 7 (sign code) must be either 0x002B or 0x002D.
- The contents of halfwords 0 to 6 must be in the range 0x0030 to 0x0039.

National decimal values having a sign code of 0x002B are interpreted as positive values.

National decimal values having a sign code of 0x002D are interpreted as negative values.

For each integer value i from 0 to 23, do the following.

The contents of nibble element i of VSR[VRT+32] are set to 0x0.

For each integer value i from 0 to 6, do the following.

The contents of nibble 3 of halfword element i of src are placed into nibble element i+24 of VSR[VRT+32].

For PS=0, the contents of nibble element 31 (i.e., sign code) of VSR[VRT+32] are set to 0xC for positive values and to 0xD for negative values.

For PS=1, the contents of nibble element 31 (i.e., sign code) of VSR[VRT+32] are set to 0xF for positive values and to 0xD for negative values.

CR field 6 is set to reflect src compared to zero.

If src is an invalid encoding of a national decimal value, the contents of VSR[VRT+32] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

<table>
<thead>
<tr>
<th>Register Data Layout for bcdcfn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
</tr>
<tr>
<td>result</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRB</th>
<th>PS</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>VSR[VRB+32]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Decimal Convert From Zoned VX-form

Let \( \text{src} \) be the zoned decimal value in \( \text{VSR}[\text{VRB}+32] \).
\( \text{src} \) is placed in \( \text{VSR}[\text{VRT}+32] \) in packed decimal format.

When \( \text{PS}=0 \), do the following.

A valid encoding of a zoned decimal value requires the following.

- The contents of bits 0:3 of byte 15 (sign code) can be any value in the range \( 0x0 \) to \( 0xF \).
- The contents of bits 0:3 of bytes 0 to 14 must be the value \( 0x3 \).
- The contents of bytes 4:7 of bytes 0 to 15 must be a value in the range \( 0x0 \) to \( 0x9 \).

Zoned decimal values having a sign code of \( 0x0, 0x1, 0x2, 0x3, 0x8, 0x9, 0xA, or 0xB \) are interpreted as positive values.

Zoned decimal values having a sign code of \( 0xA, 0xE, 0x7, 0xC, 0xD, 0x5, 0x6, \) or \( 0xF \) are interpreted as negative values.

When \( \text{PS}=1 \), do the following.

A valid encoding of a zoned decimal source operand requires the following.

- The contents of bits 0:3 of byte 15 (sign code) must be a value in the range \( 0xA \) to \( 0xF \).
- The contents of bits 0:3 of bytes 0 to 14 must be the value \( 0xF \).
- The contents of bytes 4:7 of bytes 0 to 15 must be a value in the range \( 0x0 \) to \( 0x9 \).

Zoned decimal source operands having a sign code of \( 0x0, 0x3, 0x8, 0x9, 0xA, \) or \( 0xB \) are interpreted as positive values.

Zoned decimal source operands having a sign code of \( 0xA, 0xC, 0xE, \) or \( 0xF \) are interpreted as negative values.

Positive packed decimal results are returned with a sign code of \( 0xC \).

Negative packed decimal results are returned with a sign code of \( 0xD \).

For each integer value \( i \) from 0 to 14,

- The contents of nibble element \( i \) of \( \text{VSR}[\text{VRT}+32] \) are set to \( 0x0 \).

For each integer value \( i \) from 0 to 15,

- The contents of nibble 1 of byte element \( i \) of \( \text{src} \) are placed into nibble element \( i+15 \) of \( \text{VSR}[\text{VRT}+32] \).

CR field 6 is set to reflect \( \text{src} \) compared to zero.

If \( \text{src} \) is an invalid encoding of a zoned decimal value, the contents of \( \text{VSR}[\text{VRT}+32] \) are undefined and CR field 6 is set to \( 0b00001 \).

Special Registers Altered:
CR field 6
Register Data Layout for bcdcfz.

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>
Decimal Convert To National VX-form

src is placed into VSR[VRT+32] in national decimal format.

A valid encoding of a signed packed decimal value requires the following.

- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal values with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal values with sign codes of 0xB or 0xD are interpreted as negative values.

Values greater in magnitude than $10^7 - 1$ are too large to be represented in national decimal format.

For each integer value $i$ from 0 to 6, do the following.

- The value 0x003 is placed into nibbles 0:2 of halfword element $i$ of VSR[VRT+32].
- The contents of nibble element $i+24$ of VSR[VRT+32] are placed into nibble 3 of halfword element $i$ of VSR[VRT+32].

The contents of nibble element $i+24$ of VSR[VRT+32] are set to 0x002D for positive values and to 0x002B for negative values.

CR field 6 is set to reflect src compared to zero, including whether or not src is too large to be represented in national decimal format.

If src is an invalid encoding of a packed decimal value, the contents of VSR[VRT+32] are undefined and CR field 6 is set to 0b00001.

Special Registers Altered:

CR field 6

<table>
<thead>
<tr>
<th>Register Data Layout for bcdctn.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>src</strong></td>
</tr>
<tr>
<td><strong>result</strong></td>
</tr>
</tbody>
</table>

Let src be the packed decimal value in VSR[VRT+32].
**Decimal Convert To Zoned VX-form**

```
bc dctz. VRT,VRB,PS

if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← (VSR[VRB+32].nibble[31] < 0xA)
do i = 0 to 30
  inv_flag ← inv_flag | (VSR[VRB+32].nibble[i] > 0x9)
end

ov_flag ← 0
do i = 0 to 15
  ov_flag ← ov_flag | (VSR[VRB+32].nibble[i] != 0x0)
end

src_sign ← (VSR[VRB+32].nibble[31] = 0xB) | (VSR[VRB+32].nibble[31] = 0xD)

eq_flag ← (VSR[VRB+32].nibble[0:30] = 0)
lt_flag ← eq_flag & (src_sign=1)
gt_flag ← eq_flag & (src_sign=0)
do i = 0 to 14
  result.byte[i].nibble[0] ← (PS=0) ? 0x3 : 0xF
  result.byte[i].nibble[1] ← VSR[VRB+32].nibble[i+15]
end

if src.sign=0 then
  result.byte[15].nibble[0] ← (PS=0) ? 0x3 : 0xC
else
  result.byte[15].nibble[0] ← (PS=0) ? 0x7 : 0xD
end

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ov_flag
```

Let `src` be the packed decimal value in `VSR[VRB+32]`.

`src` is placed into `VSR[VRT+32]` in zoned decimal format.

A valid encoding of a signed packed decimal value requires the following.

- The contents of nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal values with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal values with sign codes of 0xB or 0xD are interpreted as negative values.

Values greater in magnitude than \(10^{16} - 1\) are too large to be represented in zoned decimal format.

For `PS=0`, do the following.

The leftmost nibble of each digit 0-14 of the zoned decimal result is set to 0x3.

Positive zoned decimal results are returned with a sign code of 0x3.

Negative zoned decimal results are returned with a sign code of 0x7.

For `PS=1`, do the following.

The leftmost nibble of each digit 0-14 of the zoned decimal result is set to 0xF.

Positive zoned decimal results are returned with a sign code of 0xC.

Negative zoned decimal results are returned with a sign code of 0xD.

For each integer value `i` from 0 to 15, do the following.

The rightmost nibble of each digit `i` of the zoned decimal result is set to the contents of nibble `i+15` of `src`.

The result is placed into `VSR[VRT+32]`.

CR field 6 is set to reflect `src` compared to zero, including whether or not `src` is too large to be represented in zoned decimal format.

If `src` is an invalid encoding of a packed decimal value, the contents of `VSR[VRT+32]` are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

- CR field 6

---

**Register Data Layout for bcdctz.**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>127</td>
</tr>
</tbody>
</table>

---

**Version 3.1**

Power ISA™ I
Decimal Convert From Signed Quadword VX-form

Let \( \text{src} \) be the signed integer value in \( \text{VSR}[\text{VRB}+32] \).
\( \text{src} \) is placed into \( \text{VSR}[\text{VRT}+32] \) in signed packed decimal format.

For \( \text{PS}=0 \), the contents of nibble element 31 (i.e., sign code) of \( \text{VSR}[\text{VRT}+32] \) are set to \( 0xC \) for values greater than or equal to 0 and to \( 0xD \) for values less than 0.

For \( \text{PS}=1 \), the contents of nibble element 31 (i.e., sign code) of \( \text{VSR}[\text{VRT}+32] \) are set to \( 0xF \) for values greater than or equal to 0 and to \( 0xD \) for values less than 0.

If the signed integer value in \( \text{VSR}[\text{VRB}+32] \) is greater than \( 10^{31}-1 \) or less than \( -10^{31}-1 \), the value is too large to be represented in packed decimal format, and the contents of \( \text{VSR}[\text{VRT}+32] \) are undefined.

CR field 6 is set to reflect \( \text{src} \) compared to zero and whether or not \( \text{src} \) is too large in magnitude to be represented in packed decimal format.

Special Registers Altered:
CR field 6

### Register Data Layout for \texttt{bcdfsq}.

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
**Decimal Convert To Signed Quadword VX-form**

Let `src` be the packed decimal value in `VSR[VRB+32]`. `src` is placed into `VSR[VRT+32]` in signed integer format.

A valid encoding of a signed packed decimal value requires the following.

- The contents of nibble 31 (sign code) must be a value in the range `0xA` to `0xF`.
- The contents of each nibble, 0-30, must be a value in the range `0x0` to `0x9`.

Packed decimal values with sign codes of `0xA`, `0xC`, `0xE`, or `0xF` are interpreted as positive values.

Packed decimal values with sign codes of `0xB` or `0xD` are interpreted as negative values.

CR field 6 is set to reflect `src` compared to zero.

If `src` is an invalid encoding of a packed decimal value, the contents of `VSR[VRT+32]` are undefined and CR field 6 is set to `0b0001`.

**Special Registers Altered:**
CR field 6

---

### Register Data Layout for `bcdctsq`

<table>
<thead>
<tr>
<th></th>
<th>VSR[VRB+32]</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>result</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Let `inv_flag` be `(VSR[VRT+32].nibble[31] < 0xA)
inv_flag ← inv_flag | (VSR[VRB+32].nibble[i] > 0x9)
end

**Let `src_sign` be the packed decimal value in VSR[VRB+32].** `src` is placed into `VSR[VRT+32]` in signed integer format.

**A valid encoding of a signed packed decimal value requires the following.**

- The contents of nibble 31 (sign code) must be a value in the range `0xA` to `0xF`.
- The contents of each nibble, 0-30, must be a value in the range `0x0` to `0x9`.

Packed decimal values with sign codes of `0xA`, `0xC`, `0xE`, or `0xF` are interpreted as positive values.

Packed decimal values with sign codes of `0xB` or `0xD` are interpreted as negative values.

CR field 6 is set to reflect `src` compared to zero.

If `src` is an invalid encoding of a packed decimal value, the contents of `VSR[VRB+32]` are undefined and CR field 6 is set to `0b0001`.

**Special Registers Altered:**
CR field 6

---

### Register Data Layout for `bcdctsq`

<table>
<thead>
<tr>
<th></th>
<th>VSR[VRB+32]</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>result</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

486 Power ISA™ I
Vector Multiply-by-10 Unsigned Quadword VX-form

```
vmul10uq VRT,VRA

4 6 VRT 11 VRA 16 /// 21 513
```

if MSR.VEC=0 then Vector_Unavailable();

```
src ← EXTZ(VSR[VRA+32])
prod ← (src << 3) + (src << 1)
VSR[VRT+32] ← CHOP128(prod)
```

Let src be the unsigned integer value in VSR[VRA+32].

The rightmost 128 bits of the product of src multiplied by the value 10 are placed into VSR[VRT+32].

Special Registers Altered:
None

Register Data Layout for vmul10uq & vmul10cuq

```
src VSR[VRA+32]
result VSR[VRT+32]
```

Vector Multiply-by-10 & write Carry-out Unsigned Quadword VX-form

```
vmul10cuq VRT,VRA

4 6 VRT 11 VRA 16 /// 21 31
```

if MSR.VEC=0 then Vector_Unavailable();

```
src ← EXTZ(VSR[VRA+32])
prod ← (src << 3) + (src << 1)
VSR[VRT+32] ← CHOP128(prod >> 128)
```

Let src be the unsigned integer value in VSR[VRA+32].

The product of src multiplied by the value 10 is shifted right by 128 bits. The rightmost 128 bits of the shifted result is placed into VSR[VRT+32].

Special Registers Altered:
None
### Vector Multiply-by-10 Extended Unsigned Quadword VX-form

**vmul10euq**

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>577</th>
</tr>
</thead>
</table>

| 0 | 6 | 11 | 16 | 21 | 31 |

- If MSR.VEC=0 then Vector_Unavailable()

```
src ← EXTZ(VSR[VRA+32])
cin ← EXTZ(VSR[VRB+32].bit[124:127])
prod ← (src << 3) + (src << 1) + cin
VSR[VRT+32] ← CHOP128(prod)
```

Let src be the unsigned integer value in VSR[VRA+32].

Let cin be the unsigned packed decimal value in bits 124:127 of VSR[VRB+32]. Values of cin greater than 9 are undefined.

The rightmost 128 bits of the sum of cin and the product of src multiplied by the value 10 are placed into VSR[VRT+32].

**Special Registers Altered:**

None

### Register Data Layout for vmul10euq & vmul10ecuq

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

### Vector Multiply-by-10 Extended & write Carry-out Unsigned Quadword VX-form

**vmul10ecuq**

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>65</th>
</tr>
</thead>
</table>

| 0 | 6 | 11 | 16 | 21 | 31 |

- If MSR.VEC=0 then Vector_Unavailable()

```
src ← EXTZ(VSR[VRA+32])
cin ← EXTZ(VSR[VRB+32].bit[124:127])
prod ← (src << 3) + (src << 1) + cin
VSR[VRT+32] ← CHOP128(prod>>128)
```

Let src be the unsigned integer value in VSR[VRA+32].

Let cin be the unsigned packed decimal value in bits 124:127 of VSR[VRB+32]. Values of cin greater than 9 are undefined.

The sum of cin and the product of src multiplied by the value 10 is shifted right by 128 bits. The rightmost 128 bits of the shifted result is placed into VSR[VRT+32].

**Special Registers Altered:**

None
6.15.3 Decimal Integer Sign Manipulation Instructions

**Decimal Copy Sign VX-form**

The decimal value in VSR[VRA+32] is placed into VSR[VRT+32] with the sign code of the decimal value in VSR[VRB+32].

CR field 6 is set to reflect the result compared to zero.

If either the decimal value in VSR[VRA+32] or the decimal value in VSR[VRB+32] is an invalid encoding, the contents of VSR[VRT+32] are undefined and CR field 6 is set to 0b00001.

**Special Registers Altered:**
- CR field 6

---

**Register Data Layout for bcdcpsgn.**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
**Decimal Set Sign VX-form**

Let \( src \) be the packed decimal value in \( VSR[VRT+32] \).

Packed decimal values with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal values with sign codes of 0xB or 0xD are interpreted as negative values.

If \( src \) is negative, \( src \) is placed into \( VSR[VRT+32] \) with the sign code set to 0xD.

If \( src \) is positive and \( PS=0 \), \( src \) is placed into \( VSR[VRT+32] \) with the sign code set to 0xC.

If \( src \) is positive and \( PS=1 \), \( src \) is placed into \( VSR[VRT+32] \) with the sign code set to 0xF.

CR field 6 is set to reflect \( src \) compared to zero.

If \( src \) is an invalid encoding of a packed decimal value, the contents of \( VSR[VRT+32] \) are undefined and CR field 6 is set to 0b00001.

Special Registers Altered:
- CR field 6

**Register Data Layout for bcdsetsgn.**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>127</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>385</td>
</tr>
</tbody>
</table>

Version 3.1
### 6.15.4 Decimal Integer Shift and Round Instructions

#### Decimal Shift VX-form

Let $n$ be the signed integer value in byte element 7 of $VSR[VRA+32]$.

Let $src$ be the signed packed decimal value in $VSR[VRB+32]$.

A valid encoding of a signed packed decimal value requires the following.
- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal source operands with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal source operands with sign codes of 0xB or 0xD are interpreted as negative values.

If $n$ is greater than zero, $src$ is shifted left $n$ digits. Zeros are supplied to vacated digits on the right. If any non-zero digits are shifted out, an overflow occurs.

If $n$ is less than zero, $src$ is shifted right $-n$ digits. Zeros are supplied to vacated digits on the left.

If the packed decimal value in $VSR[VRB+32]$ is negative, the sign code of the result is set to 0b1101.

If the packed decimal value in $VSR[VRB+32]$ is positive, the sign code of the result is set to 0b1100 if $PS=0$ and is set to 0b1111 if $PS=1$.

The shifted result is placed into $VSR[VRT+32]$.

CR field 6 is set to reflect $src$ compared to zero, including whether or not significant digits were shifted out when the shift count is positive (i.e., left shift operation).

If $src$ is an invalid encoding of a packed decimal value, the contents of $VSR[VRT+32]$ are undefined and CR field 6 is set to 0b0001.

**Special Registers Altered:**
- CR field 6

#### Register Data Layout for bcds.

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

Let $n$ be the signed integer value in byte element 7 of $VSR[VRA+32]$.

Let $src$ be the signed packed decimal value in $VSR[VRB+32]$.

A valid encoding of a signed packed decimal value requires the following.
- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal source operands with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal source operands with sign codes of 0xB or 0xD are interpreted as negative values.

If $n$ is greater than zero, $src$ is shifted left $n$ digits. Zeros are supplied to vacated digits on the right. If any non-zero digits are shifted out, an overflow occurs.

If $n$ is less than zero, $src$ is shifted right $-n$ digits. Zeros are supplied to vacated digits on the left.

If the packed decimal value in $VSR[VRB+32]$ is negative, the sign code of the result is set to 0b1101.

If the packed decimal value in $VSR[VRB+32]$ is positive, the sign code of the result is set to 0b1100 if $PS=0$ and is set to 0b1111 if $PS=1$.

The shifted result is placed into $VSR[VRT+32]$.

CR field 6 is set to reflect $src$ compared to zero, including whether or not significant digits were shifted out when the shift count is positive (i.e., left shift operation).

If $src$ is an invalid encoding of a packed decimal value, the contents of $VSR[VRT+32]$ are undefined and CR field 6 is set to 0b0001.

**Special Registers Altered:**
- CR field 6

#### Register Data Layout for bcds.

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
Let \( n \) be the signed integer value in byte element 7 of \( \text{VSR}[\text{VRA+32}] \).

Let \( \text{src} \) be the unsigned packed decimal value in \( \text{VSR}[\text{VRB+32}] \).

A valid encoding of an unsigned packed decimal value requires the contents of each nibble 0-31 must be a value in the range 0x0 to 0x9.

If \( n \) is greater than zero, \( \text{src} \) is shifted left \( n \) digits. Zeros are supplied to vacated digits on the right. If any non-zero digits are shifted out, an overflow occurs.

If \( n \) is less than zero, \( \text{src} \) is shifted right \( -n \) digits. Zeros are supplied to vacated digits on the left.

The shifted result is placed into \( \text{VSR}[\text{VRT+32}] \).

CR field 6 is set to reflect \( \text{src} \) compared to zero, including whether or not significant digits were shifted out when the shift count is positive (i.e., left shift operation).

If \( \text{src} \) is an invalid encoding of a packed decimal value, the contents of \( \text{VSR}[\text{VRT+32}] \) are undefined and CR field 6 is set to 0b00001.

**Special Registers Altered:**
CR field 6

---

**Register Data Layout for bcdus.**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VSR[VRT+32]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VSR[VRB+32]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>31</th>
<th>129</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>VRT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>VRA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>VRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>129</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Decimal Shift & Round VX-form

**bcdsr.** VRT, VRA, VRB, PS

Let \( n \) be the signed integer value in byte element 7 of VSR\([\text{VRA+32}]\).

Let \( src \) be the signed packed decimal value in VSR\([\text{VRB+32}]\).

A valid encoding of a signed packed decimal source operand requires the following:
- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal source operands with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal source operands with sign codes of 0xB or 0xD are interpreted as negative values.

If \( n \) is greater than zero, \( src \) is shifted left \( n \) digits. Zeros are supplied to vacated digits on the right. If any non-zero digits are shifted out, an overflow occurs.

If \( n \) is less than zero, \( src \) is shifted right \(-n\) digits. Zeros are supplied to vacated digits on the left. If the value of the last nibble shifted out on the right was greater than or equal to 5, the magnitude of the result is incremented by 1.

If \( src \) is negative, the sign code of the result is set to 0b1101.

If \( src \) is positive, the sign code of the result is set to 0b1100 if \( PS=0 \) and is set to 0b1111 if \( PS=1 \).

The shifted and rounded result is placed into VSR\([\text{VRT+32}]\).

CR field 6 is set to reflect \( src \) compared to zero, including whether or not significant digits were shifted out when the shift count is positive (i.e., left shift operation).

If \( src \) is an invalid encoding of a packed decimal value, the contents of VSR\([\text{VRT+32}]\) are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:
CR field 6

**Register Data Layout for bcdsr.**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>result</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

Chapter 6. Vector Facility 493
6.15.5 Decimal Integer Truncate Instructions

Decimal Truncate VX-form

Let \(\text{length}\) be the integer value in bits 48:63 of \(\text{VSR}[\text{VRA+32}]\).

Let \(\text{src}\) be the signed decimal value in \(\text{VSR}[\text{VRB+32}]\).

A valid encoding of a packed decimal source operand requires the following.

- The contents of nibble 31 (sign code) must be a value in the range \(0xA\) to \(0xF\).
- The contents of each nibble 0-30 must be a value in the range \(0x0\) to \(0x9\).

Packed decimal values with sign codes of \(0xA\), \(0xC\), \(0xE\), or \(0xF\) are interpreted as positive values.

Packed decimal values with sign codes of \(0xB\) or \(0xD\) are interpreted as negative values.

If \(\text{src}\) is negative, the sign code of the result is set to \(0b1101\).

If \(\text{src}\) is positive, the sign code of the result is set to \(0b1110\) if \(\text{PS}=0\) and is set to \(0b1111\) if \(\text{PS}=1\).

\(\text{src}\) is copied into \(\text{VSR}[\text{VRT+32}]\) with the leftmost \(\text{length}\) digits set to \(0b0000\). If any of the leftmost \(\text{length}\) digits of the signed decimal value in \(\text{VSR}[\text{VRB+32}]\) are non-zero, an overflow occurs.

CR field 6 is set to reflect \(\text{src}\) compared to zero, including whether or not significant digits were truncated.

If \(\text{src}\) is an invalid encoding of a packed decimal value, the contents of \(\text{VSR}[\text{VRT+32}]\) are undefined and CR field 6 is set to \(0b0001\).

Special Registers Altered:
CR field 6

Register Data Layout for \(\text{bcdtrunc}\).

- \(\text{src1}\)
- \(\text{src2}\)
- \(\text{result}\)
### Decimal Unsigned Truncate VX-form

#### bcdutrunc. VRT,VRA,VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>321</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>23</td>
</tr>
</tbody>
</table>

- **inv_flag** ← 0
- **do i = 0 to 31**
  - **inv_flag** ← **inv_flag** | (VSR[VRB+32].nibble[i] > 0x9)
  - **end**
- **length** ← VSR[VRA+32].bit[48:63]
- ox_flag ← 0
- eq_flag ← (VSR[VRB+32].nibble[0:31] = 0)
- gt_flag ← (VSR[VRB+32].nibble[0:31] != 0)

- **if length < 32 then do**
  - **do i = 0 to 31-length**
    - **if** VSR[VRB+32].nibble[i] != 0b0000 **then ox_flag** ← 1
    - **result.nibble[i]** ← 0b0000
  - **end**
- **if length > 0 then do**
  - **do i = 32-length to 31**
    - **result.nibble[i]** ← VSR[VRB+32].nibble[i]
  - **end**
- **end**
- **else result** ← VSR[VRB+32]

- **VSR[VRT+32]** ← **inv_flag** ? undefined : **result**

- **CR.bit[56]** ← 0b0
- **CR.bit[57]** ← **inv_flag** ? 0b0 : **gt_flag**
- **CR.bit[58]** ← **inv_flag** ? 0b0 : **eq_flag**
- **CR.bit[59]** ← **inv_flag** | ox_flag

---

Let **length** be the integer value in bits 48:63 of VSR[VRA+32].

Let **src** be the unsigned decimal value in VSR[VRB+32].

A valid encoding of a packed decimal source operand requires the contents of each nibble 0-31 must be a value in the range 0x0 to 0x9.

**src** is copied into VSR[VRT+32] with the leftmost 32-length digits each set to 0b0000. If any of the leftmost 32-length digits of the signed decimal value in VSR[VRB+32] are non-zero, an overflow occurs.

CR field 6 is set to reflect **src** compared to zero, including whether or not significant digits were truncated.

If **src** is an invalid encoding of a packed decimal value, the contents of VSR[VRT+32] are undefined and CR field 6 is set to 0b0001.

**Special Registers Altered:**

- **CR field 6**

---

**Register Data Layout for bcdutrunc.**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td></td>
<td>VSR[VRA+32]</td>
</tr>
<tr>
<td>src2</td>
<td></td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>result</td>
<td></td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
6.16 Vector Status and Control Register Instructions

**Move To Vector Status and Control Register VX-form**

\[ \text{mtvscr} \quad \text{VRB} \]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>15</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[ \text{VSCR} \leftarrow \text{VSR[VRB+32].word[3]} \]

The contents of word element 3 of VSR[VRB+32] are placed into the VSCR.

**Special Registers Altered:**
None

**Register Data Layout for mtvscr**

<table>
<thead>
<tr>
<th>src</th>
<th>unused</th>
<th>VSR[VRB+32].word[3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**Move From Vector Status and Control Register VX-form**

\[ \text{mfvscr} \quad \text{VRT} \]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[ \text{VSR[VRT+32]} \leftarrow \text{EXTZ128}(\text{VSCR}) \]

The contents of the VSCR are placed into word element 3 of VSR[VRT+32].

The remaining word elements in VSR[VRT+32] are set to 0.

**Special Registers Altered:**
None

**Register Data Layout for mfvscr**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000_0000</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
Chapter 7. Vector-Scalar Extension Facility

7.1 Introduction

7.1.1 Overview of the Vector-Scalar Extension

Vector-Scalar Extension (VSX) provides facilities supporting vector and scalar binary floating-point operations. The following VSX features are provided to increase opportunities for vectorization.

- A unified register file, a set of Vector-Scalar Registers (VSR), supporting both scalar and vector operations is provided, eliminating the overhead of vector-scalar data transfer through storage.

- Support for word-aligned storage accesses for both scalar and vector operations is provided.

- Robust support for IEEE-754 for both vector and scalar floating-point operations is provided.

7.1.1.1 Combining the Floating-Point Registers (FPR) defined in Chapter 4. Floating-Point Facility and the Vector Registers (VR) defined in Chapter 6. Vector Facility provides additional registers to support more aggressive compiler optimizations for both vector and scalar operations. Compatibility with Floating-Point and Decimal Floating-Point Operations

The instruction sets defined in Chapter 4. Floating-Point Facility and Chapter 5. Decimal Floating-Point Facility retain their definition with one primary difference. The FPRs are mapped to doubleword element 0 of VSRs 0-31. The contents of doubleword 1 of the VSR corresponding to a source FPR specified by an instruction are ignored. The contents of doubleword 1 of a VSR corresponding to the target FPR specified by an instruction are set to 0.

7.1.1.2 Compatibility with Vector Operations

The instruction set defined in Chapter 6. Vector Facility, retains its definition with one primary difference. The VRs are mapped to VSRs 32-63.

Programming Note

Application binary interfaces extended to support VSX require special care of vector data written to VSRs 0-31 (i.e., VSRs corresponding to FPRs). Legacy scalar function calls employ doubleword-based loads and stores to preserve the contents of any nonvolatile registers. This has the adverse effect of not preserving the contents of doubleword 1 of these VSRs.

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
7.2 VSX Registers

7.2.1 Vector-Scalar Registers

Sixty-four 128-bit VSRs are provided. See Figure 105. All VSX floating-point computations and other data manipulation are performed on data residing in Vector-Scalar Registers, and results are placed into a VSR.

Depending on the instruction, the contents of a VSR are interpreted as a sequence of equal-length elements (words or doublewords) or as a quadword. Each of the elements is aligned within the VSR, as shown in Figure 105. Many instructions perform a given operation in parallel on all elements in a VSR.

Depending on the instruction, a word element can be interpreted as a signed integer word (SW), an unsigned integer word (UW), a logical mask value (MW), or a single-precision floating-point value (SP); a doubleword element can be interpreted as a doubleword signed integer (SD), a doubleword unsigned integer (UD), a doubleword mask (DM), or a double-precision floating-point value (DP). In the instructions descriptions, phrases like signed integer word element are used as shorthand for word element, interpreted as a signed integer.

Load and Store instructions are provided that transfer a byte, a specified number of bytes (up to 16), a halfword, a word, a doubleword, or a quadword between storage and a VSR, or an octword between storage and a pair of VSRs.

![Figure 105: Vector-Scalar Registers](image)

![Figure 106: Vector-Scalar Register Elements](image)

7.2.1.1 Floating-Point Registers

Chapter 4. Floating-Point Facility provides 32 64-bit FPRs. Chapter 5. Decimal Floating-Point also employs FPRs in decimal floating-point (DFP) operations. When VSX is implemented, the 32 FPRs are mapped to doubleword 0 of VSRs 0-31. For example, FPR[0] is located in doubleword element 0 of VSR[0], FPR[1] is located in doubleword element 0 of VSR[1], and so forth.

All instructions that operate on an FPR are redefined to operate on doubleword element 0 of the corresponding VSR. The contents of doubleword element 1 of the VSR corresponding to a source FPR or FPR pair for these instructions are ignored and the contents of doubleword element 1 of the VSR corresponding to the target FPR or FPR pair for these instructions are set to 0.
Figure 107. Floating-Point Registers as part of VSRs
7.2.1.2 Vector Registers

Chapter 6, Vector Facility, provides 32 128-bit VRs. When VSX is implemented, the 32 VRs are mapped to VSRs 32-63. For example, VR[0] is located in VSR[32], VR[1] is located in VSR[33], and so forth. All instructions that operate on a VR are redefined to operate on the corresponding VSR.

![Figure 108. Vector Registers as part of VSRs](image-url)
7.2.1.3 VSX Accumulators

Eight 512-bit Accumulators (ACC) are provided. Each ACC contains four 128-bit rows.

\[
\begin{align*}
\text{ACC}[i][0] &= \text{ACC}[i].\text{bit}[0:127] \\
\text{ACC}[i][1] &= \text{ACC}[i].\text{bit}[128:255] \\
\text{ACC}[i][2] &= \text{ACC}[i].\text{bit}[256:383] \\
\text{ACC}[i][3] &= \text{ACC}[i].\text{bit}[384:511] \\
\end{align*}
\]

Each ACC is associated with four VSRs in the following manner.

\[
\begin{align*}
\text{ACC}[0][0] &\leftrightarrow \text{VSR}[0] \\
\text{ACC}[0][1] &\leftrightarrow \text{VSR}[1] \\
\text{ACC}[0][2] &\leftrightarrow \text{VSR}[2] \\
\text{ACC}[0][3] &\leftrightarrow \text{VSR}[3] \\
\text{ACC}[1][0] &\leftrightarrow \text{VSR}[4] \\
\text{ACC}[1][1] &\leftrightarrow \text{VSR}[5] \\
\text{ACC}[1][2] &\leftrightarrow \text{VSR}[6] \\
\text{ACC}[1][3] &\leftrightarrow \text{VSR}[7] \\
&\vdots \\
\text{ACC}[7][0] &\leftrightarrow \text{VSR}[28] \\
\text{ACC}[7][1] &\leftrightarrow \text{VSR}[29] \\
\text{ACC}[7][2] &\leftrightarrow \text{VSR}[30] \\
\text{ACC}[7][3] &\leftrightarrow \text{VSR}[31] \\
\end{align*}
\]

While the ACCs are treated as separate registers from the VSRs, ACC[i] may use its associated VSRs 4×i to 4×i+3 as scratch space. That is, when ACC[i] contains defined data, the contents of VSRs 4×i to 4×i+3 are undefined until either a VSX Move From ACC instruction is used to copy the contents of ACC[i] to VSRs 4×i to 4×i+3 or some other instruction directly writes to one of these VSRs.

Any instruction that targets any VSR(s) associated with an ACC causes any subsequent use of that ACC as a source operand to be undefined, but causes the contents of all VSRs associated with the ACC to be undefined.

**Programming Note**

Application software must strictly adhere to the programming model described in this section to guarantee compatibility with future versions of the architecture.

For this version of the architecture, the hardware implementation provides the effect of ACC[i] and VSRs 4×i to 4×i+3 logically containing the same data. Being subject to change in future versions of this architecture, application software must not rely on this behavior. However, system software that handles context save/restore operations need only save and restore data from and to the VSRs (that is, the most current data between the VSRs and associated ACCs will be provided by the hardware implementation). The Accumulators introduce no new logical state at this time. However, future versions of the architecture may define new architectural state or re-define the backing state of the ACC registers. In turn, this may require changes to the system software to support programs written according to this version of the architecture.

The following instructions can be used to copy the contents of an ACC into its associated VSRs.

**VSX Move From Accumulator (xxmfacc)**

The contents of the source ACC are copied into the VSRs associated with the target ACC.

The following instructions can be used to initialize the contents of an ACC.

**VSX Move To Accumulator (xxmtacc)**

The contents of the VSRs associated with the target ACC are copied into the target ACC.

**VSX Set ACC to Zero (xxsetaccz)**

The target ACC is set to 0.

**[Prefixed Masked] VSX Vector 4-bit Signed Integer GER (rank-8) ([pm]xvi4ger8)**

The sum of the eight outer products of the 4-bit signed integer values in the two vector source operands are placed into the target ACC.

**[Prefixed Masked] VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4) ([pm]xvi8ger4)**

The sum of the four outer products of the 8-bit signed and unsigned integer values in the two vector source operands are placed into the target ACC.
[Prefixed Masked] VSX Vector 16-bit Signed Integer GER (rank-2) ([pm]xvi16ger2s)
The sum of the two outer products of the 16-bit signed integer values in the two vector source operands are placed into the target ACC.

[Prefixed Masked] VSX Vector 16-bit Floating-Point GER (rank-2) ([pm]xvf16ger2)
The sum of the two outer products of the 16-bit floating-point values in the two vector source operands are placed into the target ACC.

[Prefixed Masked] VSX Vector bfloat16 GER (rank-2) ([pm]xvb16ger2)
The sum of the two outer products of the bfloat16 values in the two vector source operands are placed into the target ACC.

[Prefixed Masked] VSX Vector 32-bit Floating-Point GER (rank-1) ([pm]xvf32ger)
The outer product of the 32-bit floating-point values in the two vector source operands is placed into the target ACC.

[Prefixed Masked] VSX Vector 64-bit Floating-Point GER (rank-1) ([pm]xvf64ger)
The outer product of the 64-bit floating-point values in the two vector source operands is placed into the target ACC.
7.2.2 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) controls the handling of floating-point exceptions and records status resulting from the floating-point operations. Bits 0:19 and 32:55 are status bits. Bits 56:63 are control bits.

The exception status bits in the FPSCR (bits 35:44, 53:55) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an \texttt{mcrfs}, \texttt{mtfsfi}, \texttt{mtfsf}, or \texttt{mtfsb0} instruction. The exception summary bits in the FPSCR (FX, FEX, and VX, which are bits 32:34) are not considered to be “exception status bits”, and only FX is sticky.

FX and VX are simply the ORs of other FPSCR bits. Therefore these two bits are not listed among the FPSCR bits affected by the various instructions.

The bit definitions for the FPSCR are as follows.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:28</td>
<td>Decimal Floating-Point Rounding Control (DRN)</td>
</tr>
<tr>
<td></td>
<td>This field is not used by VSX instructions.</td>
</tr>
<tr>
<td>32</td>
<td>Floating-Point Exception Summary (FX)</td>
</tr>
<tr>
<td></td>
<td>Every floating-point instruction, except \texttt{mtfsfi} and \texttt{mtfsf}, implicitly sets FX to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. \texttt{mcrfs}, \texttt{mtfsfi}, \texttt{mtfsf}, \texttt{mtfsb0}, and \texttt{mtfsb1} can alter FX explicitly.</td>
</tr>
</tbody>
</table>

\begin{center}
\textbf{Programming Note}

FX is defined not to be altered implicitly by \texttt{mtfsfi} and \texttt{mtfsf} because permitting these instructions to alter FX implicitly can cause a paradox. An example is an \texttt{mtfsfi} or \texttt{mtfsf} instruction that supplies 0 for FX and 1 for OX, and is executed when OX=0. See also the Programming Notes with the definition of these two instructions.
\end{center}

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>Floating-Point Enabled Exception Summary (FEX)</td>
</tr>
<tr>
<td></td>
<td>This bit is the OR of all the floating-point exception bits masked by their respective enable bits. \texttt{mcrfs}, \texttt{mtfsfi}, \texttt{mtfsf}, \texttt{mtfsb0}, and \texttt{mtfsb1} cannot alter FEX explicitly.</td>
</tr>
</tbody>
</table>

\begin{center}
\textbf{Programming Note}

Access to \texttt{Move To FPSCR} and \texttt{Move From FPSCR} instructions requires FP=1.
\end{center}

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>Floating-Point Invalid Operation Exception Summary (VX)</td>
</tr>
<tr>
<td></td>
<td>This bit is the OR of all the Invalid Operation exception bits. \texttt{mcrfs}, \texttt{mtfsfi}, \texttt{mtfsf}, \texttt{mtfsb0}, and \texttt{mtfsb1} cannot alter VX explicitly.</td>
</tr>
<tr>
<td>35</td>
<td>Floating-Point Overflow Exception (OX)</td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic, VSX Vector Floating-Point Arithmetic, VSX Scalar DP-SP Conversion or VSX Vector DP-SP Conversion class instruction causes an Overflow exception. See Section 7.4.3, “Floating-Point Overflow Exception” on page 542.</td>
</tr>
<tr>
<td></td>
<td>This bit can be set to 0 or 1 by a \texttt{Move To FPSCR} class instruction.</td>
</tr>
<tr>
<td>36</td>
<td>Floating-Point Underflow Exception (UX)</td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic, VSX Vector Floating-Point Arithmetic, VSX Scalar DP-SP Conversion or VSX Vector DP-SP Conversion class instruction causes an Underflow exception. See Section 7.4.4, “Floating-Point Underflow Exception” on page 548.</td>
</tr>
<tr>
<td></td>
<td>This bit can be set to 0 or 1 by a \texttt{Move To FPSCR} class instruction.</td>
</tr>
<tr>
<td>37</td>
<td>Floating-Point Zero Divide Exception (ZX)</td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic or VSX Vector Floating-Point Arithmetic class instruction causes a Zero Divide exception. See Section 7.4.2, “Floating-Point Zero Divide Exception” on page 539.</td>
</tr>
<tr>
<td></td>
<td>This bit can be set to 0 or 1 by a \texttt{Move To FPSCR} class instruction.</td>
</tr>
<tr>
<td>38</td>
<td>Floating-Point Inexact Exception (XX)</td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic, VSX Vector Floating-Point Arithmetic, VSX Scalar Integer Conversion, VSX Vector Integer Conversion, VSX Scalar Round to Floating-Point Integer, or VSX Vector Round to Floating-Point Integer class instruction causes an Inexact exception. See Section 7.4.5, “Floating-Point Inexact Exception” on page 554.</td>
</tr>
<tr>
<td></td>
<td>This bit can be set to 0 or 1 by a \texttt{Move To FPSCR} class instruction.</td>
</tr>
</tbody>
</table>
### Bit(s) Definition

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td><strong>Floating-Point Invalid Operation Exception</strong></td>
</tr>
<tr>
<td></td>
<td>(SNAN) <em>(VXSNAN)</em></td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Floating-Point and VSX Vector Floating-Point class instruction causes an SNaN type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 527. This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
</tr>
<tr>
<td>40</td>
<td><strong>Floating-Point Invalid Operation Exception</strong></td>
</tr>
<tr>
<td></td>
<td>(Inf-Inf) <em>(VXSI)</em></td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic and VSX Vector Floating-Point Arithmetic class instruction causes an Infinity – Infinity type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 527. This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
</tr>
<tr>
<td>41</td>
<td><strong>Floating-Point Invalid Operation Exception</strong></td>
</tr>
<tr>
<td></td>
<td>(Inf+Inf) <em>(VXIDI)</em></td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic and VSX Vector Floating-Point Arithmetic class instruction causes an Infinity + Infinity type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 527. This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
</tr>
<tr>
<td>42</td>
<td><strong>Floating-Point Invalid Operation Exception</strong></td>
</tr>
<tr>
<td></td>
<td>(Zero+Zero) <em>(VXZDZ)</em></td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic and VSX Vector Floating-Point Arithmetic class instruction causes a Zero + Zero type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 527. This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
</tr>
<tr>
<td>43</td>
<td><strong>Floating-Point Invalid Operation Exception</strong></td>
</tr>
<tr>
<td></td>
<td>(Inf×Zero) <em>(VXIZ</em>)</td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic and VSX Vector Floating-Point Arithmetic class instruction causes an Infinity × Zero type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 527. This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
</tr>
<tr>
<td>44</td>
<td><strong>Floating-Point Invalid Operation Exception</strong></td>
</tr>
<tr>
<td></td>
<td>(Invalid Compare) <em>(VXVC)</em></td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Compare Double-Precision, VSX Vector Compare Double-Precision, or VSX Vector Compare Single-Precision class instruction causes an Invalid Compare type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 527. This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
</tr>
<tr>
<td>45</td>
<td><strong>Floating-Point Fraction Rounded</strong> <em>(FR)</em></td>
</tr>
<tr>
<td></td>
<td>This bit is set to 0 or 1 by VSX Scalar Floating-Point Arithmetic, VSX Scalar Integer Conversion, and VSX Scalar Round to Floating-Point Integer class instructions to indicate whether or not the fraction was incremented during rounding. See Section 7.3.2.6, “Rounding” on page 518. This bit is not sticky.</td>
</tr>
<tr>
<td>46</td>
<td><strong>Floating-Point Fraction Inexact</strong> <em>(FI)</em></td>
</tr>
</tbody>
</table>
|        | This bit is set to 0 or 1 by VSX Scalar Floating-Point Arithmetic, VSX Scalar Integer Conversion, and VSX Scalar Round to Floating-Point Integer class instructions to indicate whether or not the rounded result is inexact or the instruction caused a disabled Overflow exception. See Section 7.3.2.6 on page 518. This bit is not sticky. See the definition of XX, above, regarding the relationship between FI and XX.
Bit(s) Definition

47:51 Floating-Point Result Flags (FPRF)
VSX Scalar Floating-Point Arithmetic, VSX Scalar DP-SP Conversion, VSX Scalar Convert Integer to Double-Precision, and VSX Scalar Round to Double-Precision Integer class instructions set this field based on the result placed into the target register and on the target precision, except that if any portion of the result is undefined then the value placed into FPRF is undefined.

For VSX Scalar Convert Double-Precision to Integer class instructions, the value placed into FPRF is undefined.

Additional details are as follows.

<table>
<thead>
<tr>
<th>Result Flags</th>
<th>Result Value Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>C FL FG FE FU</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>1 0 0 0 1</td>
<td>Infinity</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>Normalized Number</td>
</tr>
<tr>
<td>1 1 0 0 0</td>
<td>Denormalized Number</td>
</tr>
<tr>
<td>1 0 0 1 0</td>
<td>Zero</td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>+ Zero</td>
</tr>
<tr>
<td>1 0 1 0 0</td>
<td>+ Denormalized Number</td>
</tr>
<tr>
<td>0 0 1 0 0</td>
<td>+ Normalized Number</td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>+ Infinity</td>
</tr>
</tbody>
</table>

Table 2. Floating-Point Result Flags

47 Floating-Point Result Class Descriptor (C)
VSX Scalar Floating-Point Arithmetic, VSX Scalar DP-SP Conversion, VSX Scalar Convert Integer to Double-Precision, and VSX Scalar Round to Double-Precision Integer class instructions set this bit with the FPCC bits, to indicate the class of the result as shown in Table 2, “Floating-Point Result Flags,” on page 505. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

48 Floating-Point Less Than or Negative (FL)

49 Floating-Point Greater Than or Positive (FG)

50 Floating-Point Equal or Zero (FE)

51 Floating-Point Unordered or NaN (FU)

52 Reserved

53 Floating-Point Invalid Operation Exception (Software-Defined Condition) (VXSOFT)
This bit can be altered only by mcrfs, mtfsfi, mtfsf, mtfsb0, or mtfsb1. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 527.

Programming Note
VXSOFT can be used by software to indicate the occurrence of an arbitrary, software-defined, condition that is to be treated as an Invalid Operation exception. For example, the bit could be set by a program that computes a base 10 logarithm if the supplied input is negative.

54 Floating-Point Invalid Operation Exception (Invalid Square Root) (VXSQRT)
This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic or VSX Vector Floating-Point Arithmetic class instruction causes an Invalid Square Root type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 527.

This bit can be set to 0 or 1 by a Move To FPSCR class instruction.
<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Definition</th>
<th>Bit(s)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td><strong>Floating-Point Invalid Operation Exception (Invalid Integer Convert)</strong> (<em>VXCVI</em>)</td>
<td>60</td>
<td><strong>Floating-Point Inexact Exception Enable</strong> (<em>XE</em>)</td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Convert Double-Precision to Integer, VSX Vector Convert Double-Precision to Integer, or</td>
<td></td>
<td>This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Inexact</td>
</tr>
<tr>
<td></td>
<td>VSX Vector Convert Single-Precision to Integer class instruction causes a invalid Integer Convert type Invalid Operation</td>
<td></td>
<td>exceptions. See Section 7.4.5 , “Floating-Point Inexact Exception” on page 554.</td>
</tr>
<tr>
<td></td>
<td>exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 527.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>This bit can be set to 0 or 1 by a <em>Move To FPSCR</em> class instruction.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td><strong>Floating-Point Invalid Operation Exception Enable</strong> (<em>EI</em>)</td>
<td>61</td>
<td><strong>Floating-Point Non-IEEE Mode</strong> (<em>NI</em>)</td>
</tr>
<tr>
<td></td>
<td>This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Invalid</td>
<td></td>
<td>Floating-point non-IEEE mode is optional. If floating-point non-IEEE mode is not implemented, this bit is treated as reserved,</td>
</tr>
<tr>
<td></td>
<td>Operation exceptions. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 527.</td>
<td></td>
<td>and the remainder of the definition of this bit does not apply.</td>
</tr>
<tr>
<td>57</td>
<td><strong>Floating-Point Overflow Exception Enable</strong> (<em>OE</em>)</td>
<td></td>
<td>If floating-point non-IEEE mode is implemented, this bit has the following meaning.</td>
</tr>
<tr>
<td></td>
<td>This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Overflow</td>
<td></td>
<td>0 The processor is not in floating-point non-IEEE mode (i.e., all floating-point operations conform to the IEEE standard).</td>
</tr>
<tr>
<td></td>
<td>exceptions. See Section 7.4.3 , “Floating-Point Overflow Exception” on page 542.</td>
<td></td>
<td>1 The processor is in floating-point non-IEEE mode.</td>
</tr>
<tr>
<td>58</td>
<td><strong>Floating-Point Underflow Exception Enable</strong> (<em>UE</em>)</td>
<td></td>
<td>When the processor is in floating-point non-IEEE mode, the remaining FPSCR bits is permitted to have meanings different from</td>
</tr>
<tr>
<td></td>
<td>This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Underflow</td>
<td></td>
<td>those given in this document, and floating-point operations need not conform to the IEEE standard. The effects of executing a</td>
</tr>
<tr>
<td></td>
<td>exceptions. See Section 7.4.4 , “Floating-Point Underflow Exception” on page 548.</td>
<td></td>
<td>given floating-point instruction with NI=1, and any additional requirements for using non-IEEE mode, are implementation-dependent.</td>
</tr>
<tr>
<td>59</td>
<td><strong>Floating-Point Zero Divide Exception Enable</strong> (<em>ZE</em>)</td>
<td></td>
<td>The results of executing a given instruction in non-IEEE mode is permitted to vary between implementations, and between different</td>
</tr>
<tr>
<td></td>
<td>This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Zero Divide</td>
<td></td>
<td>executions on the same implementation.</td>
</tr>
<tr>
<td></td>
<td>exceptions. See Section 7.4.2 , “Floating-Point Zero Divide Exception” on page 539.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Programming Note**

When the processor is in floating-point non-IEEE mode, the results of floating-point operations is permitted to be approximate, and performance for these operations might be better, more predictable, or less data-dependent than when the processor is not in non-IEEE mode. For example, in non-IEEE mode an implementation is permitted to return 0 instead of a denormalized number and return a large number instead of an infinity.
Floating-Point Rounding Control \( (R\|n) \)

This field is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions that round their result and the rounding mode is not implied by the opcode.

This bit can be explicitly set or reset by a new Move To FPSCR class instruction.

See Section 7.3.2.6, “Rounding” on page 518.

<table>
<thead>
<tr>
<th>Value</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>01</td>
<td>Round toward Zero</td>
</tr>
<tr>
<td>10</td>
<td>Round toward +Infinity</td>
</tr>
<tr>
<td>11</td>
<td>Round toward -Infinity</td>
</tr>
</tbody>
</table>
7.3 VSX Operations

7.3.1 VSX Floating-Point Arithmetic Overview

This section describes the floating-point arithmetic and exception model supported by Vector-Scalar Extension. Except for extensions to support 32-bit single-precision floating-point vector operations, the models are identical to that described in Chapter 4. Floating-Point Facility.

The processor (augmented by appropriate software support, where required) implements a floating-point system compliant with the ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic* (hereafter referred to as the IEEE standard). That standard defines certain required "operations" (addition, subtraction, and so on). Herein, the term, floating-point operation, is used to refer to one of these required operations and to additional operations defined (e.g., those performed by Multiply-Add or Reciprocal Estimate instructions). A Non-IEEE mode is also provided. This mode, which is permitted to produce results not in strict compliance with the IEEE standard, allows shorter latency.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in VSRs; to move floating-point data between storage and these registers.

These instructions are divided into two categories.

- computational instructions

  The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. There are two forms of computational instructions, scalar, which perform a single floating-point operation, and vector, which perform either two double-precision floating-point operations or four single-precision operations. Computational instructions place status information into the Floating-Point Status and Control Register. They are the instructions described in Sections 7.6.1.3 through 7.6.1.8.2.

- noncomputational instructions

  The noncomputational instructions are those that perform loads and stores, move the contents of a VSR to another floating-point register possibly altering the sign, and select the value from one of two VSRs based on the value in a third VSR. The operations performed by these instructions are not considered floating-point operations. These instructions do not alter the Floating-Point Status and Control Register. They are the instructions listed in Sections 7.6.1.1, 7.6.1.2.1, and 7.6.1.3 through .

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number \(2^{\text{exponent}}\). Encodings are provided in the data format to represent finite numeric values, \(\pm\infty\), and values that are “Not a Number” (NaN). Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. NaNs might be used to indicate such things as uninitialized variables and can be produced by certain invalid operations.

There is one class of exceptional events that occur during instruction execution that is unique to Vector-Scalar Extension and Floating-Point: the Floating-Point Exception. Floating-point exceptions are signaled with bits set in the FPSCR. They can cause the system floating-point enabled exception error handler to be invoked, precisely or imprecisely, if the proper control bits are set.

**Floating-Point Exceptions**

The following floating-point exceptions are detected by the processor:

- Invalid Operation exception (VX)
  - SNaN (VXSNaN)
  - Infinity- Infinity (VXI1S1)
  - Infinity+Infinity (VXI1D1)
  - Zero+Zero (VXZDZ)
  - Infinity×Zero (VXI1M2)
  - Invalid Compare (VXXC)
  - Software-Defined Condition (VXSOFT)
  - Invalid Square Root (VXSQRT)
  - Invalid Integer Convert (VXXVI)

- Zero Divide exception (ZX)
- Overflow exception (OX)
- Underflow exception (UX)
- Inexact exception (XX)

Each floating-point exception, and each category of Invalid Operation exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. See Section 7.2.2, “Floating-Point Status and Control Register” on page 503 for a description of these exception and enable bits, and Section 7.3.3, “VSX Floating-Point Execution Models” on page 521 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.
7.3.2 VSX Floating-Point Data

7.3.2.1 Data Format

This architecture defines the representation of a floating-point value in five different binary fixed-length formats, 16-bit half-precision format, 16-bit bfloat16 format, 32-bit single-precision format, 64-bit double-precision format, and 128-bit quad-precision format. The half-precision format is used for half-precision floating-point data in storage and registers. The bfloat16 format is used for bfloat16 floating-point data in storage and registers. The single-precision format is used for single-precision floating-point data in storage and registers. The double-precision format is used for double-precision floating-point data in storage and registers. The quad-precision format is used for quad-precision floating-point data in storage and registers.

The lengths of the exponent and the fraction fields differ between these five formats. The structure of the half-precision, bfloat16, single-precision, double-precision, and quad-precision formats is shown in Figure 109, Figure 110, Figure 111, Figure 112, and Figure 113, respectively.

Values in floating-point format are composed of three fields:

- **S** sign bit
- **EXP** exponent+bias
- **FRACTION** fraction

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized (subnormal) numbers or zero and is located in the unit bit position (that is, the first bit to the left of the binary point). Values representable within the three floating-point formats can be specified by the parameters listed in Table 3.

---

**Figure 109. Binary floating-point half-precision format (binary16)**

<table>
<thead>
<tr>
<th>S</th>
<th>EXP</th>
<th>FRACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
</tr>
</tbody>
</table>

**Figure 110. Binary floating-point bfloat16 format (bfloat16)**

<table>
<thead>
<tr>
<th>S</th>
<th>EXP</th>
<th>FRACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
</tr>
</tbody>
</table>

**Figure 111. Binary floating-point single-precision format (binary32)**

<table>
<thead>
<tr>
<th>S</th>
<th>EXP</th>
<th>FRACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

**Figure 112. Binary floating-point double-precision format (binary64)**

<table>
<thead>
<tr>
<th>S</th>
<th>EXP</th>
<th>FRACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63</td>
</tr>
</tbody>
</table>

**Figure 113. Binary floating-point quad-precision format (binary128)**

<table>
<thead>
<tr>
<th>S</th>
<th>EXP</th>
<th>FRACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>bfloat16</td>
<td>binary16</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>Exponent Bias</td>
<td>+127</td>
<td>+15</td>
</tr>
<tr>
<td>Maximum Exponent</td>
<td>+127</td>
<td>+15</td>
</tr>
<tr>
<td>Minimum Exponent</td>
<td>-126</td>
<td>-14</td>
</tr>
</tbody>
</table>

Widths (bits):

<table>
<thead>
<tr>
<th>Format</th>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1</td>
<td>8</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>23</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>11</td>
<td>24</td>
<td>53</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>15</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{Nmax} = (1-2^{-8}) \times 2^{128} = 3.4 \times 10^{38} \\
\text{Nmin} = 1.0 \times 2^{-126} = 1.2 \times 10^{-38} \\
\text{Dmin} = 1.0 \times 2^{-133} = 9.2 \times 10^{-41} \\
\]

\[
\text{Value is approximate} \\
\text{Smallest (in magnitude) representable denormalized number.} \\
\text{Largest (in magnitude) representable number.} \\
\text{Smallest (in magnitude) representable normalized number.} \\
\]

Table 3. Binary floating-point fields
7.3.2.2 Value Representation

This architecture defines numeric and nonnumeric values representable within each of the three supported formats. The numeric values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The nonnumeric values representable are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible however to define restricted operations among numbers and infinities as defined below. The relative location on the real number line for each of the defined entities is shown in Figure 114.

Figure 114. Approximation to real numbers

<table>
<thead>
<tr>
<th>-INF</th>
<th>-NOR</th>
<th>-DEN</th>
<th>0+0</th>
<th>+DEN</th>
<th>+NOR</th>
<th>+INF</th>
</tr>
</thead>
</table>

The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers

Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers (±NOR)

These are values that have a biased exponent value in the range:

- 1 to 30 in half-precision format
- 1 to 254 in bfloat16 format
- 1 to 254 in single-precision format
- 1 to 2046 in double-precision format
- 1 to 32766 in quad-precision format

They are values in which the implied unit bit is 1.

Normalized numbers are interpreted as follows:

\[
\text{NOR} = (-1)^s \times 2^{E} \times (1.fraction)
\]

where \( s \) is the sign, \( E \) is the unbiased exponent, and \( 1.fraction \) is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

Zero values (±0)

These are values that have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (that is, comparison regards +0 as equal to -0).

Denormalized numbers (±DEN)

These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0.

Denormalized numbers are interpreted as follows:

\[
\text{DEN} = (-1)^s \times 2^{E_{\text{min}}} \times 0.fraction
\]

where \( E_{\text{min}} \) is the minimum representable exponent value.

- 14 for half-precision
- 126 for bfloat16
- 126 for single-precision
- 1022 for double-precision
- 16382 for quad-precision.

Infinities (±INF)

These are values that have the maximum biased exponent value:

- 31 in half-precision format
- 255 in bfloat16 format
- 255 in single-precision format
- 2047 in double-precision format
- 32767 in quad-precision format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

\[-\text{Infinity} < \text{every finite number} < +\text{Infinity}\]

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 7.4.1, "Floating-Point Invalid Operation Exception" on page 527.

For comparison operations, +Infinity compares equal to +Infinity and -Infinity compares equal to -Infinity.
Not a Numbers (NaNs)

These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (that is, NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0, the NaN is a Signaling NaN; otherwise it is a Quiet NaN.

Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions.

Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation exception is disabled (VE=0). Quiet NaNs propagate through all floating-point operations except ordered comparison and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

Assume the following generic arithmetic templates.

\[
\begin{align*}
f(src1,src3,src2) & \quad \text{ex: } \text{result} = (src1 \times src3) - src2 \\
f(src1,src2) & \quad \text{ex: } \text{result} = src1 \times src2 \\
f(src1) & \quad \text{ex: } \text{result} = f(src1)
\end{align*}
\]

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a trap-disabled Invalid Operation exception, the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

\[
\begin{align*}
\text{if src1 is a NaN} & \quad \text{then result = Quiet(src1)} \\
\text{else if src2 is a NaN (if there is a src2)} & \quad \text{then result = Quiet(src2)} \\
\text{else if src3 is a NaN (if there is a src3)} & \quad \text{then result = Quiet(src3)} \\
\text{else if disabled invalid operation exception} & \quad \text{then result = generated QNaN}
\end{align*}
\]

where \(\text{Quiet}(x)\) means \(x\) if \(x\) is a QNaN and \(x\) converted to a QNaN if \(x\) is an SNaN. Any instruction that generates a QNaN as the result of a disabled Invalid Operation exception generates the value,

- \(0x7E00\) for half-precision results,
- \(0x7FC0\) for bfloat16 results,
- \(0x7FC0_{0000}\) for single-precision results,
- \(0x7FF8_{0000_0000_0000}\) for double-precision results,
- \(0x7FFF_{8000_0000_0000_0000_0000_0000_0000}\) for quad-precision results.

Note that the M-form multiply-add-type instructions use the \(B\) source operand to specify \(src3\) and the \(T\) target operand to specify \(src2\), whereas A-form multiply-add-type instructions use the \(B\) source operand to specify \(src2\) and the \(T\) target operand to specify \(src3\).

A double-precision NaN is considered to be representable in single-precision format if and only if the low-order 29 bits of the double-precision NaN’s fraction are zero.

7.3.2.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same signs, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation \(x-y\) is the same as the sign of the result of the add operation \(x+(-y)\).
When the sum of two operands with opposite sign, or the difference of two operands with the same signs, is exactly zero, the sign of the result is positive in all rounding modes except Round toward -Infinity, in which mode the sign is negative.

- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.

- The sign of the result of a Square Root or Reciprocal Square Root Estimate operation is always positive, except that the square root of -0 is -0 and the reciprocal square root of -0 is -Infinity.

- The sign of the result of a Convert From Integer or Round to Floating-Point Integer operation is the sign of the operand being converted.

For the Multiply-Add instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

### 7.3.2.4 Normalization and Denormalization

The intermediate result of an arithmetic instruction can require normalization and/or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or rounding instruction produces an intermediate result which carries out of the significand, or in which the significand is nonzero but has a leading zero bit, it is not a normalized number and must be normalized before it is stored. For the carry-out case, the significand is shifted right one bit, with a one shifted into the leading significand bit, and the exponent is incremented by one. For the leading-zero case, the significand is shifted left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The Guard bit and the Round bit (see Section 7.3.3.1, “VSX Execution Model for IEEE Operations” on page 521) participate in the shift with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result can have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be “Tiny” and the stored result is determined by the rules described in Section 7.4.4, “Floating-Point Underflow Exception” on page 548. These rules can require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format's minimum value. If any significant bits are lost in this shifting process, “Loss of Accuracy” has occurred (See Section 7.4.4, “Floating-Point Underflow Exception” on page 548) and Underflow exception is signaled.

### Engineering Note

When denormalized numbers are operands of multiply, divide, and square root operations, some implementations might prenormalize the operands internally before performing the operations.

### 7.3.2.5 Data Handling and Precision

Scalar double-precision floating-point data is represented in double-precision format in VSRs and storage.

Vector double-precision floating-point data is represented in double-precision format in VSRs and storage.

Scalar single-precision floating-point data is represented in double-precision format in VSRs and in single-precision format in storage.

Vector single-precision floating-point data is represented in single-precision format in VSRs and storage.

Double-precision operands may be used as input for double-precision scalar arithmetic operations.

Double-precision operands may be used as input for single-precision scalar arithmetic operations when trapping on overflow and underflow exceptions is disabled.

Single-precision operands may be used as input for double-precision and single-precision scalar arithmetic operations.

Double-precision operands may be used as input for double-precision vector arithmetic operations.

Single-precision operands may be used as input for single-precision vector arithmetic operations.

Instructions are also provided for manipulations which do not require double-precision or single-precision. In addition, instructions are provided to access an integer representation in GPRs.
Half-Precision Operands

Instructions are provided to convert between half-precision and single-precision formats for vector data in VSRs and between half-precision and double-precision formats for scalar data. Note that scalar double-precision format is identical to scalar single-precision format.

An instruction is provided to explicitly convert half-precision format operands in a VSR to single-precision format. Scalar single-precision floating-point is enabled with six types of instruction.

1. **VSX Scalar Convert Half-Precision to Double-Precision format XX2-form**

   The half-precision floating-point value in the rightmost halfword in doubleword element 0 of the source VSR is placed into the doubleword element 0 of the target VSR in double-precision format.

2. **VSX Scalar Convert with round Double-Precision to Half-Precision format XX2-form**

   The double-precision value in doubleword element 0 of the source VSR is rounded to to half-precision, checking the exponent for half-precision range and handling any exceptions according to respective enable bits, and places the result into the rightmost halfword of doubleword element 0 of the target VSR in half-precision format.

   Source operand values greater in magnitude than $2^{39}$ when Overflow is enabled ($OE=1$) produce undefined results because the value cannot be scaled into the half-precision normalized range.

   Source operand values smaller in magnitude than $2^{-38}$ when Underflow is enabled ($UE=1$) produce undefined results because the value cannot be scaled into the half-precision normalized range.

3. **VSX Vector Convert bfloat16 to Single-Precision format XX2-form**

   The bfloat16 floating-point value in the rightmost halfword of each word element of the source VSR is placed into the corresponding word element of the target VSR in single-precision format.

4. **VSX Vector Convert with round Single-Precision to bfloat16 format XX2-form**

   The single-precision floating-point value in each word element $i$ of the source VSR is rounded to bfloat16 precision and placed into the rightmost halfword of the corresponding word element of the target VSR in bfloat16 format.

bfloat16 Operands

Instructions are provided to convert between bfloat16 and single-precision formats for vector data in VSRs.

An instruction is provided to explicitly convert bfloat16 format operands in a VSR to single-precision format.

1. **VSX Vector Convert bfloat16 to Single-Precision format XX2-form**

   The bfloat16 floating-point value in the rightmost halfword of each word element of the source VSR is placed into the corresponding word element of the target VSR in single-precision format.

2. **VSX Vector Convert with round Single-Precision to bfloat16 format XX2-form**

   The single-precision floating-point value in each word element $i$ of the source VSR is rounded to bfloat16 precision and placed into the rightmost halfword of the corresponding word element of the target VSR in bfloat16 format.

Single-Precision Operands

For single-precision scalar data, a conversion from single-precision format to double-precision format is performed when loading from storage into a VSR and a conversion from double-precision format to single-precision format is performed when storing from a VSR to storage. No floating-point exceptions are caused by these instructions.

Instructions are provided to convert between single-precision and double-precision formats for scalar and vector data in VSRs.

An instruction is provided to explicitly convert a double format operand in a VSR to single-precision. Scalar single-precision floating-point is enabled with six types of instructions.

1. **Load VSX Scalar Single-Precision**

   $[p]lxssp$ and $lxssp$ access a floating-point operand in single-precision format in storage, converts it to double-precision format, and loads it into a VSR. No floating-point exceptions are caused by these instructions.

2. **VSX Scalar Round to Single-Precision XX2-form**

   $xsrsp$ rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any
exceptions according to respective enable bits, and places that operand into a VSR in double-precision format. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of \texttt{xrsps}, \texttt{xrsrp} does not alter the value. Values greater in magnitude than $2^{-319}$ when Overflow is enabled (\texttt{OE=1}) produce undefined results because the value cannot be scaled back into the normalized range. Values smaller in magnitude than $2^{-318}$ when Underflow is enabled (\texttt{UE=1}) produce undefined results because the value cannot be scaled back into the normalized range.

3. **VSX Scalar Convert Single-Precision to Double-Precision format**

**VSX Scalar Convert Single-Precision to Double-Precision format** \texttt{XX2-form (xscvspdp)} accesses a floating-point operand in single-precision format from word element \texttt{0} of the source VSR, converts it to double-precision format, and places it into doubleword element \texttt{0} of the target VSR.

**VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form (xscvspdpn)** accesses a floating-point operand in single-precision format from word element \texttt{0} of the source VSR, converts it to double-precision format, and places it into doubleword element \texttt{0} of the target VSR. \texttt{xscvspdpn} does not set any exception status (i.e., \texttt{VXSNAN}).

4. **VSX Scalar Convert Double-Precision to Single-Precision format** [Non-Signalling]

**VSX Scalar Convert with round Double-Precision to Single-Precision format** \texttt{XX2-form (xscvdpsp)} rounds the double-precision floating-point value in doubleword element \texttt{0} of the source VSR to single-precision, and places the result into word elements \texttt{0} and \texttt{1} of the target VSR in single-precision format. This function would be used to port scalar floating-point data to a format compatible for single-precision vector operations. Values greater in magnitude than $2^{-319}$ when Overflow is enabled (\texttt{OE=1}) produce undefined results because the value cannot be scaled back into the normalized range. Values smaller in magnitude than $2^{-318}$ when Underflow is enabled (\texttt{UE=1}) produce undefined results because the value cannot be scaled back into the normalized range.

**VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling**

\texttt{XX2-form (xscvdpsspn)} directly converts the single-precision floating-point value represented in double-precision format in doubleword element \texttt{0} of the source VSR to single-precision format, without rounding, and places the result into word elements \texttt{0} and \texttt{1} of the target VSR in single-precision format. \texttt{xscvdpsspn} does not set any exception status (i.e., \texttt{VXSNAN}).

5. **VSX Scalar Single-Precision Arithmetic**

This form of instruction takes operands from the VSRs in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single-precision format. Status bits, in the FPSCR and optionally in the Condition Register, are set to reflect the single-precision result. The result is then placed into the target VSR in double-precision format. The result lies in the range supported by the single format.

If any input value is not representable in single-precision format and either \texttt{OE=1} or \texttt{UE=1}, the result placed into the target VSR and the setting of status bits in the FPSCR are undefined.

For \texttt{xrsrsp} or \texttt{xrsqrtesp}, if the input value is finite and has an unbiased exponent greater than $+127$, the input value is interpreted as an Infinity.

6. **Store VSX Scalar Single-Precision**

Store VSX Scalar Single-Precision DS-form (\texttt{stxssp}), Prefixed Store VSX Scalar Single-Precision 8LS:D-form (\texttt{pstxssp}), and Store VSX Scalar Single-Precision Indexed X-form (\texttt{stxsspx}) convert a single-precision value that is in double-precision format to single-precision format and stores that operand into storage. No floating-point exceptions are caused by these instructions. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding five types.)

When the result of a Load VSX Scalar Single-Precision (\texttt{lxsspx}), a VSX Scalar Round to Single-Precision (\texttt{xrsrp}), or a VSX Scalar Single-Precision Arithmetic instruction is stored in a VSR, the low-order 29 bits of \texttt{FRACTION} are zero.

---

1. VSX Scalar Single-Precision Arithmetic instructions:
   \texttt{xssaddsp}, \texttt{xsvdivsp}, \texttt{xsmulp}, \texttt{xrsrp}, \texttt{xssubsp}, \texttt{xsmaddsp}, \texttt{xsmaddsp}, \texttt{xsmsubasp}, \texttt{xsmsubmsp}, \texttt{xsmnaddsp}, \texttt{xsmnaddsp}, \texttt{xsnmsubsp}, \texttt{xsmnmsubsp}
Integer-Valued Operands

Instructions are provided to round floating-point operands to integer values in floating-point format. To facilitate exchange of data between the floating-point and integer processing, instructions are provided to convert between floating-point double and single-precision format and integer word and doubleword format in a VSR. Computation on integer-valued operands can be performed using arithmetic instructions of the required precision. (The results might not be integer values.) The three groups of instructions provided specifically to support integer-valued operands are described below.

1. Rounding to a floating-point integer

   **VSX Scalar Round to Double-Precision Integer**: Instructions round a double-precision operand to an integer value in double-precision format. These instructions can also be used for single-precision operands represented in double-precision format.
VSX Vector Round to Double-Precision Integer\(^1\) instructions round each double-precision vector operand element to an integer value in double-precision format.

VSX Vector Round to Single-Precision Integer\(^2\) instructions round each single-precision vector operand element to an integer value in single-precision format.

Except for \texttt{xsrdpic}, \texttt{xvrdpic}, and \texttt{xvrspic}, rounding is performed using the rounding mode specified by the opcode. For \texttt{xsrdpic}, \texttt{xvrdpic}, and \texttt{xvrspic}, rounding is performed using the rounding mode specified by \texttt{Rn}.

VSX Round to Floating-Point Integer\(^3\) instructions can cause Invalid Operation (\texttt{VXSNAN}) exceptions.

\texttt{xsrdpic}, \texttt{xvrdpic}, and \texttt{xvrspic} can also cause Inexact exception.

See Sections 7.3.2.6 and 7.3.3.1 for more information about rounding.

2. Converting floating-point format to integer format

VSX Scalar Double-Precision to Integer Format Conversion\(^4\) instructions convert a double-precision operand to 32-bit or 64-bit signed or unsigned integer format. These instructions can also be used for single-precision operands represented in double-precision format.

VSX Vector Double-Precision to Integer Format Conversion\(^5\) instructions convert either double-precision or single-precision vector operand elements to 32-bit or 64-bit signed or unsigned integer format.

VSX Vector Single-precision to Integer Doubleword Format Conversion\(^6\) instructions converts the single-precision value in each odd-numbered word element of the source vector operand to a 64-bit signed or unsigned integer format.

VSX Vector Single-precision to Integer Word Format Conversion\(^7\) instructions converts the single-precision value in each word element of the source vector operand to either a 32-bit signed or unsigned integer format.

Rounding is performed using Round Towards Zero rounding mode. These instructions can cause Invalid Operation (\texttt{VXSNAN}, \texttt{VXCVI}) and Inexact exceptions.

3. Converting integer format to floating-point format

VSX Scalar Integer Doubleword to Double-Precision Format Conversion\(^8\) instructions convert a 64-bit signed or unsigned integer to a double-precision floating-point value and returns the result in double-precision format.

VSX Scalar Integer Doubleword to Single-Precision Format Conversion\(^9\) instructions converts a 64-bit signed or unsigned integer to a single-precision floating-point value and returns the result in double-precision format.

VSX Vector Integer Doubleword to Double-Precision Format Conversion\(^10\) instructions converts the 64-bit signed or unsigned integer in each doubleword element in the source vector operand to double-precision floating-point format.

VSX Vector Integer Doubleword to Single-Precision Format Conversion\(^11\) instructions convert either double-precision or single-precision vector operand elements to 32-bit or 64-bit signed or unsigned integer format.

\texttt{xsrdpic}, \texttt{xvrdpic}, \texttt{xsrdpim}, \texttt{xvrdpim}, \texttt{xsrdpiz}, \texttt{xvrdpiz}.

\texttt{xvrdpic}, \texttt{xvrdpip}, \texttt{xvrdpim}, \texttt{xvrdpiz}.

\texttt{xvrdpim}.

\texttt{xvrdpiz}.

\texttt{vxcvdpuxds}, \texttt{xvscvdpuxds}, \texttt{xxcvdpuxws}, \texttt{xscvdpuxds}.

\texttt{xvrdpim}.

\texttt{xvrdpiz}.

\texttt{xvcvdpuxds}.

\texttt{xvscvdpuxds}, \texttt{xvcvdpuxws}.

\texttt{xvscvdpuxds}.

\texttt{xvcvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvcvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.

\texttt{xvscvdpuxds}.
instructions convert the 64-bit signed or unsigned integer in each doubleword element in the source vector operand to single-precision floating-point format.

VSX Vector Integer Word to Single-Precision Format Conversion\(^1\) instructions convert the 32-bit signed or unsigned integer in each word element in the source vector operand to single-precision floating-point format.

Rounding is performed using the rounding mode specified in \(\text{RN}\). Because of the limitations of the source format, only an Inexact exception can be generated.

### 7.3.2.6 Rounding

The material in this section applies to operations that have numeric operands (that is, operands that are not infinities or NaNs). Rounding the intermediate result of such an operation can cause an Overflow exception, an Underflow exception, or an Inexact exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 7.3.2.2, “Value Representation” and Section 7.4, “VSX Floating-Point Exceptions” for the cases not covered here.

The floating-point arithmetic, and rounding and conversion instructions round their intermediate results. With the exception of the estimate instructions, these instructions produce an intermediate result that can be regarded as having unbounded precision and exponent range. All but two groups of these instructions normalize or denormalize the intermediate result prior to rounding and then place the final result into the target element of the target VSR in either double-precision, single-precision, or quad-precision format.

The scalar round to double-precision integer, vector round to double-precision integer, and convert double-precision to integer instructions with biased exponents ranging from 1022 through 1074 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 1075. (Intermediate results with biased exponents 1075 or larger are already integers, and with biased exponents 1021 or less round to zero.) After rounding, the final result for vector round to single-precision integer is normalized and put in double-precision format, and, for vector convert single-precision to integer is converted to a signed or unsigned integer.

\(\text{FR}\) and \(\text{FI}\) generally indicate the results of rounding. Each of the scalar instructions which rounds its intermediate result sets these bits. There are no vector instructions that modify \(\text{FR}\) and \(\text{FI}\). If the fraction is incremented during rounding, \(\text{FR}\) is set to 1, otherwise \(\text{FR}\) is set to 0. If the result is inexact, \(\text{FI}\) is set to 1, otherwise \(\text{FI}\) is set to zero. The scalar round to double-precision integer instructions are exceptions to this rule, setting \(\text{FR}\) and \(\text{FI}\) to 0. The scalar double-precision estimate instructions set \(\text{FR}\) and \(\text{FI}\) to undefined values. The remaining scalar floating-point instructions do not alter \(\text{FR}\) and \(\text{FI}\).

Four user-selectable rounding modes are provided through the Floating-Point Rounding Control field in the FPSCR. See Section 7.2.2, “Floating-Point Status and Control Register” on page 503. These are encoded as follows.

<table>
<thead>
<tr>
<th>(\text{RN})</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>01</td>
<td>Round towards Zero</td>
</tr>
<tr>
<td>10</td>
<td>Round towards +Infinity</td>
</tr>
<tr>
<td>11</td>
<td>Round towards -Infinity</td>
</tr>
</tbody>
</table>

A fifth rounding mode is provided in the round to floating-point integer instructions (Section 7.6.1.8.2 on page 572), Round to Nearest Away.

A sixth rounding mode is provided in the quad-precision floating-point instructions, Round to Odd.

---

1. VSX Vector Integer Word to Single-Precision Format Conversion instructions: 
\[\text{xscvsxwsp, xscvuxwsp}\]
Let \( Z \) be the intermediate arithmetic result or the operand of a convert operation. If \( Z \) can be represented exactly in the target format, the result in all rounding modes is \( Z \) as represented in the target format. If \( Z \) cannot be represented exactly in the target format, let \( Z_1 \) and \( Z_2 \) bound \( Z \) as the next larger and next smaller numbers representable in the target format. Then \( Z_1 \) or \( Z_2 \) can be used to approximate the result in the target format.

Figure 115 shows the relation of \( Z \), \( Z_1 \), and \( Z_2 \) in this case. The following rules specify the rounding in the four modes.

See Section 7.3.3.1, “VSX Execution Model for IEEE Operations” on page 521 for a detailed explanation of rounding.

Figure 115 also summarizes the rounding actions for floating-point intermediate result for all supported rounding modes.

Programming Note

Round to Odd rounding mode is useful when the results of a Quad-Precision Arithmetic instruction are required to be rounded to a shorter precision while avoiding a double rounding error. In this case, the rounding mode of the Quad-Precision Arithmetic instruction is overridden as Round To Odd by setting the \( RO \) bit in the instruction encoding to 1, then the result of that Quad-Precision Arithmetic instruction can be rounded to the desired shorter precision using the rounding mode specified in \( RN \) by following with a VSX Scalar Round Quad-Precision to Double-Extended-Precision for 15-bit exponent range and 64-bit significand precision, VSX Scalar Round Quad-Precision to Double-Precision for 11-bit exponent range and 53-bit significand precision, or VSX Scalar Round Quad-Precision to Single-Precision for 8-bit exponent range and 24-bit significand precision. For example,

\[
\begin{align*}
&\text{asaddpo } Tx, A, B ; \text{ use Round to Odd override } (RO=1) \\
&\text{asrgqep } Tdxp, Tx ; \text{ final QP result rounded to DXP}
\end{align*}
\]

To return a quad-precision result rounded to double-precision requires a 3-instruction sequence,

\[
\begin{align*}
&\text{asaddpo } Tx, A, B ; \text{ use Round to Odd override } (RO=1) \\
&\text{ascvqdp } Temp, Tx ; \text{ QP result rounded & converted to DP} \\
&\text{ascdqgp } Tdp, Temp ; \text{ final QP result rounded to DP}
\end{align*}
\]

To return a quad-precision result rounded to single-precision requires a 4-instruction sequence,

\[
\begin{align*}
&\text{asaddpo } Tx, A, B ; \text{ use Round to Odd override } (RO=1) \\
&\text{ascvqdp } Temp, Tx ; \text{ QP result rounded to DP using Round to Odd & converted to DP format} \\
&\text{asrsr } Temp, Temp ; \text{ DP result is rounded to SP} \\
&\text{ascvdpq } Tsp, Temp ; \text{ final QP result rounded to SP}
\end{align*}
\]
Round to Nearest Away
Choose Z if Z is representable in the target precision.

Otherwise, choose the value that is closer to Z (Z₁ or Z₂). In case of a tie, choose the one that is furthest away from 0.

Round to Nearest Even
Choose Z if Z is representable in the target precision.

Otherwise, choose the value that is closer to Z (Z₁ or Z₂). In case of a tie, choose the one that is even (least significant bit is 0).

Round to Odd
Choose Z if Z is representable in the target precision.

Otherwise, choose the value (Z₁ or Z₂) that is odd (least significant bit is 1).

Round toward Zero
Choose Z if Z is representable in the target precision.

Otherwise, choose the smaller in magnitude (Z₁ or Z₂).

Round toward +Infinity
Choose Z if Z is representable in the target precision.

Otherwise, choose Z₁.

Round toward -Infinity
Choose Z if Z is representable in the target precision.

Otherwise, choose Z₂.

Figure 115.Selection of Z₁ and Z₂
7.3.3 VSX Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (that is, operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 7.3.2.2 and Section 7.3.3 for the cases not covered here.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow and underflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

- Underflow during multiplication using a denormalized operand.
- Overflow during division using a denormalized divisor.
- Undeflow during division using denormalized dividend and a large divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands.

VSX defines both scalar and vector double-precision floating-point operations to operate only on double-precision operands. VSX also defines vector single-precision floating-point operations to operate only on single-precision operands.

7.3.3.1 VSX Execution Model for IEEE Operations

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:p-1 comprise the significand of the intermediate result (where p is the length of the significand).

![Figure 116.IEEE quad-precision (binary128) floating-point execution model (p=113)](image)

![Figure 117.IEEE double-extended-precision floating-point execution model (p=64)](image)

![Figure 118.IEEE double-precision (binary64) floating-point execution model (p=53)](image)

![Figure 119.IEEE single-precision (binary32) floating-point execution model (p=24)](image)

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

For the quad-precision execution model, FRACTION is a 112-bit field that accepts the fraction of the operand.

For the double-extended-precision execution model, FRACTION is a 63-bit field that accepts the fraction of the operand. This model is used only by the VSX Scalar Round to Double-Extended-Precision instruction.

For the double-precision execution model, FRACTION is a 52-bit field that accepts the fraction of the operand.

For the single-precision execution model, FRACTION is a 23-bit field that accepts the fraction of the operand.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low-order bits of the accumulator to provide the effect of an unbounded significand. The G and R bits are required for postnormalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that appear to the low-order side of the R bit, resulting from either shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. Table 4 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number.
next lower in magnitude (NL), and the representable number next higher in magnitude (NH).

<table>
<thead>
<tr>
<th>G</th>
<th>R</th>
<th>X</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>IR is exact</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>IR closer to NL</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>IR midway between NL and NH</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>IR closer to NH</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Interpretation of G, R, and X bits

Table 5 shows the positions of the Guard, Round, and Sticky bits for quad-precision, double-extended precision, double-precision and single-precision floating-point numbers relative to the accumulator illustrated in Figures 116, 117, 118, and 119.

<table>
<thead>
<tr>
<th>Format</th>
<th>Guard</th>
<th>Round</th>
<th>Sticky</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>G bit</td>
<td>R bit</td>
<td>X bit</td>
</tr>
<tr>
<td>Single</td>
<td>24</td>
<td>25</td>
<td>OR of bits 26:52, G, R, X</td>
</tr>
</tbody>
</table>

Table 5. Location of the Guard, Round, and Sticky bits in the IEEE execution model

The significand of the intermediate result is prepared for rounding by shifting its contents right, if required, until the least significant bit to be retained is in the low-order bit position of the fraction.

Six rounding modes are provided as described in Section 7.3.2.6, “Rounding” on page 518. The rules for rounding in each mode are as follows.

- **Round to Nearest Even**
  - If IR is exact, choose IR.
  - Otherwise, if IR is closer to NL, choose NL.
  - Otherwise, if IR is closer to NH, choose NH.
  - Otherwise, if IR is midway between NL and NH, choose whichever of NL and NH is even.

- **Round to Nearest Away**
  - If IR is exact, choose IR.
  - Otherwise, if G=0, choose NL.
  - Otherwise, if G=1, choose NH.

- **Round to Odd**
  - If IR is exact, choose IR.
  - Otherwise, choose NL, and if G=1, R=1, or X=1, the least-significant bit of the result is set to 1.

Four of the rounding modes are user-selectable through RN.

<table>
<thead>
<tr>
<th>RN</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>0b01</td>
<td>Round toward Zero</td>
</tr>
<tr>
<td>0b10</td>
<td>Round toward +Infinity</td>
</tr>
<tr>
<td>0b11</td>
<td>Round toward -Infinity</td>
</tr>
</tbody>
</table>

Round to Nearest Away is provided in the VSX Round to Floating-Point Integer instructions (Section 7.6.1.8.2 on page 572).

Round to Odd is provided in the VSX Quad-Precision Floating-Point Arithmetic instructions as an override to the rounding mode selected by RN with the rules for rounding as follows.

If G=1, R=1, or X=1, the result is inexact.

If rounding results in a carry into C, the significand is shifted right one position and the exponent is incremented by one. This yields an inexact result, and possibly also exponent overflow. Fraction bits are stored to the target VSR.

7.3.3.2 VSX Execution Model for Multiply-Add Type Instructions

This architecture provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar, except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the
following format, where bits 0:106 comprise the significand of the intermediate result.

<table>
<thead>
<tr>
<th>S</th>
<th>C</th>
<th>L</th>
<th>FRACTION</th>
<th>X'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>106</td>
</tr>
</tbody>
</table>

**Figure 120. Multiply-add 64-bit execution model**

The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the FRACTION and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the FRACTION) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount that is added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the X' bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 6 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

<table>
<thead>
<tr>
<th>Format</th>
<th>Guard</th>
<th>Round</th>
<th>Sticky</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>53</td>
<td>54</td>
<td>OR of 55:105, X'</td>
</tr>
<tr>
<td>Single</td>
<td>24</td>
<td>25</td>
<td>OR of 26:105, X'</td>
</tr>
</tbody>
</table>

**Table 6. Location of the Guard, Round, and Sticky bits in the multiply-add execution model**

The rules for rounding the intermediate result are the same as those given in Section 7.3.3.1.

If the instruction is a negative multiply-add or negative multiply-subtract type instruction, the final result is negated.
### 7.4 VSX Floating-Point Exceptions

This architecture defines the following floating-point exceptions under the IEEE-754 exception model:

- Invalid Operation exception

  - SNaN
  - Infinity
  - Infinity×Infinity
  - Zero
  - Infinity×Zero
  - Invalid Compare
  - Software-Defined Condition
  - Invalid Square Root
  - Invalid Integer Convert

- Zero Divide exception
- Overflow exception
- Underflow exception
- Inexact exception

These exceptions, other than Invalid Operation exception resulting from a Software-Defined Condition, can occur during execution of computational instructions. An Invalid Operation exception resulting from a Software-Defined Condition occurs when a Move To FPSCR instruction sets VXSOFT to 1.

Each floating-point exception, and each category of Invalid Operation exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates the occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 525), whether and how the system floating-point enabled exception error handler is invoked. In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow exception depends on the setting of the enable bit.

A single instruction, other than `mtfsfi` or `mtfsf`, can set more than one exception bit only in the following cases:

- An Inexact exception can be set with an Overflow exception.
- An Inexact exception can be set with an Underflow exception.
- An Invalid Operation exception (SNaN) is set with an Invalid Operation exception (Infinity×0) for multiply-add class instructions for which the values being multiplied are infinity and zero and the value being added is an SNaN.
- An Invalid Operation exception (SNaN) can be set with an Invalid Operation exception (Invalid Compare) for ordered comparison instructions.
- An Invalid Operation exception (SNaN) can be set with an Invalid Operation exception (Invalid Integer Convert) for convert to integer instructions.

When an exception occurs, the writing of a result to the target register can be suppressed, or a result can be delivered, depending on the exception.

The writing of a result to the target register is suppressed for the certain kinds of exceptions, based on whether the instruction is a vector or a scalar instruction, so that there is no possibility that one of the operands is lost. For other kinds of exceptions and also depending on whether the instruction is a vector or a scalar instruction, a result is generated and written to the destination specified by the instruction causing the exception. The result can be a different value for the enabled and disabled conditions for some of these exceptions. Table 7 lists the types of exceptions and indicates whether a result is written to the target VSR or suppressed.

<table>
<thead>
<tr>
<th>On exception type...</th>
<th>Scalar Instruction Results</th>
<th>Vector Instruction Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled Invalid Operation</td>
<td>suppressed</td>
<td>suppressed</td>
</tr>
<tr>
<td>Enabled Zero Divide</td>
<td>suppressed</td>
<td>suppressed</td>
</tr>
<tr>
<td>Enabled Overflow</td>
<td>written</td>
<td>suppressed</td>
</tr>
<tr>
<td>Enabled Underflow</td>
<td>written</td>
<td>suppressed</td>
</tr>
<tr>
<td>Enabled Inexact</td>
<td>written</td>
<td>suppressed</td>
</tr>
<tr>
<td>Disabled Invalid Operation</td>
<td>written</td>
<td>written</td>
</tr>
</tbody>
</table>

Table 7. Exception Types Result Suppression
The subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of traps and trap handlers. In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the trap enabled case; the expectation is that the exception is detected by software, which revises the result. An FPSCR exception enable bit of 0 causes generation of the default result value specified for the trap disabled (or no trap occurs or trap is not implemented) case. The expectation is that the exception is not detected by software, which uses the default result. The result to be delivered in each case for each exception is described in the following sections.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is required for all exceptions, all FPSCR exception enable bits must be set to 0, and Ignore Exceptions Mode (see below) should be used. In this case, the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur: software can inspect the FPSCR exception bits, if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1, and a mode other than Ignore Exceptions Mode must be used. In this case, the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The system floating-point enabled exception error handler is also invoked if a Move To FPSCR instruction causes an exception bit and the corresponding enable bit both to be 1. The Move To FPSCR instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The location of these bits and the requirements for altering them are described in Book III. The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception. The effects of the four possible settings of these bits are as follows.

<table>
<thead>
<tr>
<th>FE0</th>
<th>FE1</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Ignore Exceptions Mode</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Imprecise Nonrecoverable Mode</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Imprecise Recoverable Mode</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Precise Mode</td>
</tr>
</tbody>
</table>

Table 7. Exception Types Result Suppression

<table>
<thead>
<tr>
<th>On exception type...</th>
<th>Scalar Instruction Results</th>
<th>Vector Instruction Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled Zero Divide</td>
<td>written</td>
<td>written</td>
</tr>
<tr>
<td>Disabled Overflow</td>
<td>written</td>
<td>written</td>
</tr>
<tr>
<td>Disabled Underflow</td>
<td>written</td>
<td>written</td>
</tr>
<tr>
<td>Disabled Inexact</td>
<td>written</td>
<td>written</td>
</tr>
</tbody>
</table>

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have been completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. The instruction at which the system floating-point enabled exception error handler is invoked has completed if it is the excepting instruction.
and there is only one such instruction. Otherwise, it has not begun execution, or has been partially executed in some cases, as described in Book III.

**Programming Note**

In any of the three non-Precise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, because of instructions initiated before the *Floating-Point Status and Control Register* instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In both Imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the system floating-point enabled exception error handler that result from instructions initiated before the *Floating-Point Status and Control Register* instruction to occur. This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.

The last sentence of the paragraph preceding this Programming Note can apply only in the Imprecise modes, or if the mode has just been changed from Ignore Exceptions Mode to some other mode. It always applies in the latter case.

To obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to 0.

- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.

- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.

- Precise Mode can degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.
7.4.1 Floating-Point Invalid Operation Exception

7.4.1.1 Definition

An Invalid Operation exception occurs when an operand is invalid for the specified operation. The invalid operations are:

- **SNaN**
  Any floating-point operation on a Signaling NaN.

- **Infinity–Infinity**
  Magnitude subtraction of infinities.

- **Infinity+Infinity**
  Floating-point division of infinity by infinity.

- **Zero÷Zero**
  Floating-point division of zero by zero.

- **Infinity × Zero**
  Floating-point multiplication of infinity by zero.

- **Invalid Compare**
  Floating-point ordered comparison involving a NaN.

- **Invalid Square Root**
  Floating-point square root or reciprocal square root of a nonzero negative number.

- **Invalid Integer Convert**
  Floating-point-to-integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN.

An Invalid Operation exception also occurs when an `mtfsfi`, `mtfsf`, or `mtfsb1` instruction is executed that sets `VXSOFT` to 1 (Software-Defined Condition).

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

7.4.1.2 Action for VE=1

When Invalid Operation exception is enabled (VE=1) and an Invalid Operation exception occurs, the following actions are taken:

For any of the following instructions,

- **VSX Scalar Floating-Point Arithmetic instructions**
- **VSX Scalar DP-SP Conversion instructions**
- **VSX Scalar Convert Floating-Point to Integer instructions**
- **VSX Scalar Round to Floating-Point Integer instructions**

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   - `VXSNAN` (if SNaN)
   - `VXISI` (if Infinity–Infinity)
   - `VXIDI` (if Infinity÷Infinity)
   - `VXZDZ` (if Zero÷Zero)
   - `VXIMZ` (if Infinity×Zero)
   - `VXSQRT` (if Invalid Square Root)
   - `VXCVI` (if Invalid Integer Convert)

2. Update of `VSR[XT]` is suppressed.

3. `FR` and `FI` are set to zero.

4. `FPRF` is unchanged.
For VSX Scalar Floating-Point Compare instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   - VXSNAN (if SNaN)
   - VXVC (if Invalid Compare)

2. FR, FI, and C are unchanged.

3. FPCC is set to reflect unordered.

For any of the following instructions,

- VSX Scalar Quad-Precision Arithmetic instructions:
  - xsaddqp[0], xsdivqp[0], xsmulpq[0], xssqrtqp[0], xssubqp[0]
  - xsmaddqp[0], xsmsubqp[0], xsnmaddqp[0], xsnmsubqp[0]

- VSX Scalar Quad-Precision Convert to Integer instructions:
  - xscvqpsdz, xscvqpswz, xscvqpudz, xscvqpuwz

- VSX Scalar Round Quad-Precision to Double-Extended-Precision (xsrqpxp)
- VSX Scalar Round to Quad-Precision Integer (xsrqpi)
- VSX Scalar Round to Quad-Precision Integer with Inexact (xsrqpix)
- VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] (xscvqdp[0])

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   - VXSNAN (if SNaN)
   - VXSI (if Infinity - Infinity)
   - VXDI (if Infinity ÷ Infinity)
   - VXZDZ (if Zero ÷ Zero)
   - VXIMZ (if Infinity × Zero)
   - VXSQRT (if Invalid Square Root)
   - VXCVI (if Invalid Integer Convert)

2. VSR[VRT+32] is not modified.

3. FR and FI are set to zero. FPRF is not modified.

For any of the following instructions,

- VSX Scalar Compare Ordered Quad-Precision (xscmpoqp)
- VSX Scalar Compare Unordered Quad-Precision (xscmpuqp)

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   - VXSNAN (if SNaN)
   - VXVC (if Invalid Compare)

2. FR, FI, and C are not modified. FPCC is set to reflect unordered.
For any of the following instructions,

VSX Scalar Convert Half-Precision to Double-Precision format (\texttt{xscvpdp})
VSX Scalar Convert with round Double-Precision to Half-Precision format (\texttt{xscvdp})

do the following.

1. \texttt{VXSN} is set to 1.
2. \texttt{VSR[XT]} is not modified.
3. \texttt{FR} and \texttt{FI} are set to 0. \texttt{FPRF} is not modified.

For any of the following instructions,

VSX Vector Convert Half-Precision to Single-Precision format (\texttt{xvchpsp})
VSX Vector Convert with round Single-Precision to Half-Precision format (\texttt{xvcsphp})

do the following.

1. \texttt{VXSN} is set to 1.
2. \texttt{VSR[XT]} is not modified.
3. \texttt{FR}, \texttt{FI}, and \texttt{FPRF} are not modified.

For any of the following instructions,

VSX Vector Floating-Point Arithmetic instructions:
VSX Vector Floating-Point Compare instructions:
VSX Vector DP-SP Conversion instructions:
VSX Vector Convert Floating-Point to Integer instructions:
VSX Vector Round to Floating-Point Integer instructions:

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   \begin{itemize}
   \item \texttt{VXSN} (if SNaN)
   \item \texttt{VXSI} (if Infinity – Infinity)
   \item \texttt{VXDI} (if Infinity ÷ Infinity)
   \item \texttt{VXZD} (if Zero ÷ Zero)
   \item \texttt{VXIM} (if Infinity × Zero)
   \item \texttt{VXVC} (if Invalid Compare)
   \item \texttt{VXSQRT} (if Invalid Square Root)
   \item \texttt{VXCVI} (if Invalid Integer Convert)
   \end{itemize}
2. Update of \texttt{VSR[XT]} is suppressed for all vector elements.
3. \texttt{FR} and \texttt{FI} are unchanged.
4. \texttt{FPRF} is unchanged.
7.4.1.3 Action for VE=0

When Invalid Operation exception is disabled (VE=0) and an Invalid Operation exception occurs, the following actions are taken:

For the VSX Scalar Convert with round Double-Precision to Single-Precision format (xscvdpsp) instruction:

1. VXSNAN is set to 1.
2. The single-precision representation of a Quiet NaN is placed into word elements 0 and 1 of VSR[XT]. The contents of word elements 2 and 3 of VSR[XT] are set to 0.
3. FR and FI are set to 0.
4. FPRF is set to indicate the class of the result (Quiet NaN).

For the VSX Vector Single-Precision Arithmetic instructions, VSX Vector Single-Precision Maximum/Minimum instructions, the VSX Vector Convert with round Double-Precision to Single-Precision format (xvcvdpsp) instruction, and the VSX Vector Round to Single-Precision Integer instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN  (if SNaN)
   VXSI    (if Infinity – Infinity)
   VXDI    (if Infinity + Infinity)
   VXZDZ   (if Zero ÷ Zero)
   VXIMZ   (if Infinity × Zero)
   VXSQRT  (if Invalid Square Root)

2. The single-precision representation of a Quiet NaN is placed into its respective word element of VSR[XT], and for xvcvdpsp, is also placed into bits 32:63 of its respective doubleword element of VSR[XT].
3. FR, FI, and FPRF are not modified.

For the VSX Scalar Double-Precision Arithmetic instructions, VSX Scalar Double-Precision Maximum/Minimum instructions, the VSX Scalar Convert Single-Precision to Double-Precision format (xscvspdp) instruction, and the VSX Scalar Round to Double-Precision Integer instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN  (if SNaN)
   VXSI    (if Infinity – Infinity)
   VXDI    (if Infinity + Infinity)
   VXZDZ   (if Zero ÷ Zero)
   VXIMZ   (if Infinity × Zero)
   VXSQRT  (if Invalid Square Root)

2. The double-precision representation of a Quiet NaN is placed into doubleword element 0 of VSR[XT].
   The contents of doubleword element 1 of VSR[XT] are set to 0.
3. FR and FI are set to 0.
4. FPRF is set to indicate the class of the result (Quiet NaN).
For any of the following instructions,

VSX Scalar Quad-Precision Arithmetic instructions:
\[
\begin{align*}
\text{xssaddqp} & \text{[o]}, \text{xssdivqp} [\text{[o]}], \text{xssmulqp} [\text{[o]}], \text{xssqrtqp} [\text{[o]}], \text{xssubqp} [\text{[o]}] \\
\text{xsmaddqp} [\text{[o]}], \text{xsmsubqp} [\text{[o]}], \text{xsnmaddqp} [\text{[o]}], \text{xsnmsubqp} [\text{[o]}]
\end{align*}
\]

VSX Scalar Quad-Precision Round to Integer (xsqrpi)
VSX Scalar Quad-Precision Round to Integer with Inexact (xsqrpix)

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

\[
\begin{align*}
\text{VXSNAN} & \quad \text{(if SNaN)} \\
\text{VXI SI} & \quad \text{(if Infinity \cdot \text{Infinity)}} \\
\text{VXI DI} & \quad \text{(if Infinity + \text{Infinity)}} \\
\text{VXZDZ} & \quad \text{(if Zero + Zero)} \\
\text{VXI MZ} & \quad \text{(if Infinity \times \text{Zero)}} \\
\text{VXSQRT} & \quad \text{(if Invalid Square Root)}
\end{align*}
\]

2. The quad-precision representation of a Quiet NaN is placed into VSR[VRT+32].

3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).

For vsx scalar round quad-precision to double-extended-precision (xsrqpxp), do the following.

1. VXSNAN is set to 1.

2. The Quiet NaN is placed into VSR[VRT+32] in quad-precision format.

3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).

For any of the following instructions,

VSX Scalar Compare Ordered Quad-Precision (xscmpoqp)
VSX Scalar Compare Unordered Quad-Precision (xscmpoqp)

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

\[
\begin{align*}
\text{VXSNAN} & \quad \text{(if SNaN)} \\
\text{VXVC} & \quad \text{(if Invalid Compare)}
\end{align*}
\]

2. FR, FI and C are unchanged. FPCC is set to reflect unordered.

For vsx scalar convert with round quad-precision to double-precision format [using round to odd] (xscvqdpd[o]), do the following.

1. VXSNAN is set to 1.

2. The double-precision Quiet NaN result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.

\[
0x0000_0000_0000_0000
\]

is placed into doubleword element 1 of VSR[VRT+32].

3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).
For VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format (xscvqpsdz), do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   VXSNAN  (if SNaN)
   VXCVI   (if Invalid Integer Convert)

2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a positive number or +Infinity.
   0x8000_0000_0000_0000 is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a negative number, -Infinity, or NaN.
   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].

3. FR and FI are set to 0. FPRF is undefined.

For VSX Scalar Convert with round to zero Quad-Precision to Signed Word format (xscvqpswz), do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   VXSNAN  (if SNaN)
   VXCVI   (if Invalid Integer Convert)

2. 0x7FFF_FFFF is placed into word element 1 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a positive number or +Infinity.
   0x8000_0000 is placed into word element 1 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a negative number, -Infinity, or NaN.
   0x0000_0000 is placed into word elements 0, 2, and 3 of VSR[VRT+32].

3. FR and FI are set to 0. FPRF is undefined.

For VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format (xscvqpudz), do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   VXSNAN  (if SNaN)
   VXCVI   (if Invalid Integer Convert)

2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a positive number or +Infinity.
   0x0000_0000_0000_0000 is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a negative number, -Infinity, or NaN.
   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].

3. FR and FI are set to 0. FPRF is undefined.
For VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format (\texttt{xscvqpuwz}), do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   \begin{itemize}
   \item \texttt{VXSNAN} \hspace{1em} (if SNaN)
   \item \texttt{VXCVI} \hspace{1em} (if Invalid Integer Convert)
   \end{itemize}

2. 0xFFFF_FFFF is placed into word element 1 of \texttt{VSR[VRT+32]} if the quad-precision operand in \texttt{VSR[VRB+32]} is a positive number or +Infinity.

   \begin{itemize}
   \item 0x0000_0000 is placed into word element 1 of \texttt{VSR[VRT+32]} if the quad-precision operand in \texttt{VSR[VRB+32]} is a negative number, -Infinity, or NaN.
   \item 0x0000_0000 is placed into word elements 0, 2, and 3 of \texttt{VSR[VRT+32]}.
   \end{itemize}

3. \texttt{FR} and \texttt{FI} are set to 0. \texttt{FPRF} is undefined.

For VSX Scalar Convert with round Double-Precision to Half-Precision format (\texttt{xscvdphp}), do the following.

1. \texttt{VXSNAN} is set to 1.

2. The half-precision representation of a Quiet NaN is placed into the rightmost halfword of doubleword element 0 of \texttt{VSR[XT]}. The contents of the leftmost 3 halfwords of doubleword element 0 of \texttt{VSR[XT]} are set to 0. The contents of doubleword element 1 of \texttt{VSR[XT]} are set to 0.

3. \texttt{FR} and \texttt{FI} are set to 0. \texttt{FPRF} is set to indicate the class of the result (Quiet NaN).

For VSX Scalar Convert Half-Precision to Double-Precision format (\texttt{xscvhpdp}), do the following.

1. \texttt{VXSNAN} is set to 1.

2. The double-precision representation of a Quiet NaN is placed into doubleword element 0 of \texttt{VSR[XT]}. The contents of doubleword element 1 of \texttt{VSR[XT]} are set to 0.

3. \texttt{FR} and \texttt{FI} are set to 0. \texttt{FPRF} is set to indicate the class of the result (Quiet NaN).

For any of the following instructions,

\begin{itemize}
\item VSX Vector Double-Precision Arithmetic instructions
\item VSX Vector Double-Precision Maximum/Minimum instructions
\item VSX Vector Convert Single-Precision to Double-Precision format (\texttt{xvcvspdp})
\item VSX Vector Round to Double-Precision Integer instructions
\end{itemize}

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   \begin{itemize}
   \item \texttt{VXSNAN} \hspace{1em} (if SNaN)
   \item \texttt{VXISI} \hspace{1em} (if Infinity – Infinity)
   \item \texttt{VXIDI} \hspace{1em} (if Infinity ÷ Infinity)
   \item \texttt{VXZDZ} \hspace{1em} (if Zero ÷ Zero)
   \item \texttt{VXIMZ} \hspace{1em} (if Infinity × Zero)
   \item \texttt{VXSQRT} \hspace{1em} (if Invalid Square Root)
   \end{itemize}

2. The double-precision representation of a Quiet NaN is placed into its respective doubleword element of \texttt{VSR[XT]}.

3. \texttt{FR}, \texttt{FI}, and \texttt{FPRF} are not modified.
For the **VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format** (**xscvdpsxd**) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN  (if SNaN)  
   VXCVI   (if Invalid Integer Convert)  

2. \texttt{0xFFF_FFFF_FFFF_FFFF} is placed into doubleword element 0 of \texttt{VSR[XT]} if the double-precision operand in doubleword element 0 of \texttt{VSR[XB]} is a positive number or +Infinity.

   \texttt{0x8000_0000_0000_0000} is placed into doubleword element 0 of \texttt{VSR[XT]} if the double-precision operand in doubleword element 0 of \texttt{VSR[XB]} is a negative number, -Infinity, or NaN.

   The contents of doubleword element 1 of \texttt{VSR[XT]} are set to 0.

3. \texttt{FR} and \texttt{FI} are set to 0.

4. FPRF is undefined.

For the **VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format** (**xscvpuxd**) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN  (if SNaN)  
   VXCVI   (if Invalid Integer Convert)  

2. \texttt{0xFFFF_FFFF_FFFF_FFFF} is placed into doubleword element 0 of \texttt{VSR[XT]} if the double-precision operand in doubleword element 0 of \texttt{VSR[XB]} is a positive number or +Infinity.

   \texttt{0x0000_0000_0000_0000} is placed into doubleword element 0 of \texttt{VSR[XT]} if the double-precision operand in doubleword element 0 of \texttt{VSR[XB]} is a negative number, -Infinity, or NaN.

   The contents of doubleword element 1 of \texttt{VSR[XT]} are set to 0.

3. \texttt{FR} and \texttt{FI} are set to 0.

4. FPRF is undefined.

For the **VSX Scalar Convert with round to zero Double-Precision to Signed Word format** (**xscvdpsxw**) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN  (if SNaN)  
   VXCVI   (if Invalid Integer Convert)  

2. \texttt{0xFFF_FFFF} is placed into word elements 0 and 1 of \texttt{VSR[XT]} if the double-precision operand in doubleword element 0 of \texttt{VSR[XB]} is a positive number or +Infinity.

   \texttt{0x8000_0000} is placed into word elements 0 and 1 of \texttt{VSR[XT]} if the double-precision operand in doubleword element 0 of \texttt{VSR[XB]} is a negative number, -Infinity, or NaN.

   The contents of word elements 2 and 3 of \texttt{VSR[XT]} are set to 0.

3. \texttt{FR} and \texttt{FI} are set to 0.

4. FPRF is undefined.
For the VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format ($xscvdpuwxw$) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   - VXSNAN  (if SNaN)
   - VXCVI  (if Invalid Integer Convert)

2. $0xFFFF_FFFF$ is placed into word elements 0 and 1 of $VSR[XT]$ if the double-precision operand in doubleword element 0 of $VSR[XB]$ is a positive number or $\pm\infty$.
   
   $0x0000_0000$ is placed into word elements 0 and 1 of $VSR[XT]$ if the double-precision operand in doubleword element 0 of $VSR[XB]$ is a negative number, $-\infty$, or NaN.
   
   The contents of word elements 2 and 3 of $VSR[XT]$ are set to 0.

3. $FR$ and $FI$ are set to 0.

4. FPRF is undefined.

For the VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format ($xvcvdpsxd$) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   - VXSNAN  (if SNaN)
   - VXCVI  (if Invalid Integer Convert)

2. $0x7FFF_FFFF_FFFF_FFFF$ is placed into doubleword element $i$ of $VSR[XT]$ if the double-precision operand in the corresponding doubleword element of $VSR[XB]$ is a positive number or $\pm\infty$.
   
   $0x8000_0000_0000_0000$ is placed into its respective doubleword element $i$ of $VSR[XT]$ if the double-precision operand in the corresponding doubleword element of $VSR[XB]$ is a negative number, $-\infty$, or NaN.

3. $FR$, $FI$, and FPRF are not modified.

For the VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format ($xvcvdpuxd$) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   - VXSNAN  (if SNaN)
   - VXCVI  (if Invalid Integer Convert)

2. $0xFFFF_FFFF_FFFF_FFFF$ is placed into doubleword element $i$ of $VSR[XT]$ if the double-precision operand in doubleword element $i$ of $VSR[XB]$ is a positive number or $\pm\infty$.
   
   $0x0000_0000_0000_0000$ is placed into doubleword element $i$ of $VSR[XT]$ if the double-precision operand in doubleword element $i$ of $VSR[XB]$ is a negative number, $-\infty$, or NaN.

3. $FR$, $FI$, and FPRF are not modified.
For the VSX Vector Convert with round to zero Double-Precision to Signed Word format (\texttt{xvcvdpsxw}) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   \begin{itemize}
   \item \texttt{VXSNAN} \hspace{1em} (if SNaN)
   \item \texttt{VXCVI} \hspace{1em} (if Invalid Integer Convert)
   \end{itemize}

2. \texttt{0xFFF_FFFF} is placed into word elements \(i \times 2\) and \(i \times 2 + 1\) of VR[XT] if the double-precision operand in doubleword element \(i\) of VR[XB] is a positive number or +Infinity.

   \texttt{0x0000_0000} is placed into word elements \(i \times 2\) and \(i \times 2 + 1\) of VR[XT] if the double-precision operand in doubleword element \(i\) of VR[XB] is a negative number, -Infinity, or NaN.

3. \texttt{FR}, \texttt{FI}, and \texttt{FPRF} are not modified.

For the VSX Vector Convert with round to zero Double-Precision to Unsigned Word format (\texttt{xvcudpuxw}) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   \begin{itemize}
   \item \texttt{VXSNAN} \hspace{1em} (if SNaN)
   \item \texttt{VXCVI} \hspace{1em} (if Invalid Integer Convert)
   \end{itemize}

2. \texttt{0xFFFF_FFFF} is placed into word elements \(i \times 2\) and \(i \times 2 + 1\) of VR[XT] if the double-precision operand in doubleword element \(i\) of VR[XB] is a positive number or +Infinity.

   \texttt{0x0000_0000} is placed into word elements \(i \times 2\) and \(i \times 2 + 1\) of VR[XT] if the double-precision operand in doubleword element \(i\) of VR[XB] is a negative number, -Infinity, or NaN.

3. \texttt{FR}, \texttt{FI}, and \texttt{FPRF} are not modified.

For the VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format (\texttt{xvcvpsxd}) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   \begin{itemize}
   \item \texttt{VXSNAN} \hspace{1em} (if SNaN)
   \item \texttt{VXCVI} \hspace{1em} (if Invalid Integer Convert)
   \end{itemize}

2. \texttt{0xFFF_FFFF_FFFF_FFFF} is placed into doubleword element \(i\) of VR[XT] if the single-precision operand in word element \(i \times 2\) of VR[XB] is a positive number or +Infinity.

   \texttt{0x8000_0000_0000_0000} is placed into doubleword element \(i\) of VR[XT] if the single-precision operand in word element \(i \times 2\) of VR[XB] is a negative number, -Infinity, or NaN.

3. \texttt{FR}, \texttt{FI}, and \texttt{FPRF} are not modified.
For the VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format (xvcvspuxd) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   \[
   \text{VXSNAN (if SNaN)}
   \]
   \[
   \text{VXCVI (if Invalid Integer Convert)}
   \]

2. \(0xFFFF_FFFF_FFFF_FFFF\) is placed into doubleword element \(i\) of \(\text{VSR}[\text{XT}]\) if the single-precision operand in word element \(i \times 2\) of \(\text{VSR}[\text{XB}]\) is a positive number or +Infinity.

   \(0x0000_0000_0000_0000\) is placed into doubleword element \(i\) of \(\text{VSR}[\text{XT}]\) if the single-precision operand in word element \(i \times 2\) of \(\text{VSR}[\text{XB}]\) is a negative number, -Infinity, or NaN.

3. \(\text{FR}, \text{FI}, \text{and FPRAF}\) are not modified.

For the VSX Vector Convert with round to zero Single-Precision to Signed Word format (xvcvpsxw) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   \[
   \text{VXSNAN (if SNaN)}
   \]
   \[
   \text{VXCVI (if Invalid Integer Convert)}
   \]

2. \(0x7FFF_FFFF\) is placed into word element \(i\) of \(\text{VSR}[\text{XT}]\) if the single-precision operand in word element \(i\) of \(\text{VSR}[\text{XB}]\) is a positive number or +Infinity.

   \(0x8000_0000\) is placed into word element \(i\) of \(\text{VSR}[\text{XT}]\) if the single-precision operand in word element \(i\) of \(\text{VSR}[\text{XB}]\) is a negative number, -Infinity, or NaN.

3. \(\text{FR}, \text{FI}, \text{and FPRAF}\) are not modified.

For the VSX Vector Convert with round to zero Single-Precision to Unsigned Word format (xvcvpsuxw) instruction, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   \[
   \text{VXSNAN (if SNaN)}
   \]
   \[
   \text{VXCVI (if Invalid Integer Convert)}
   \]

2. \(0xFFFF_FFFF\) is placed into word element \(i\) of \(\text{VSR}[\text{XT}]\) if the single-precision operand in the corresponding word element \(2 \times i\) of \(\text{VSR}[\text{XB}]\) is a positive number or +Infinity.

   \(0x0000_0000\) is placed into word element \(i\) of \(\text{VSR}[\text{XT}]\) if the single-precision operand in word element \(2 \times i\) of \(\text{VSR}[\text{XB}]\) is a negative number, -Infinity, or NaN.

3. \(\text{FR}, \text{FI}, \text{and FPRAF}\) are not modified.

For the VSX Scalar Floating-Point Compare instructions, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   \[
   \text{VXSNAN (if SNaN)}
   \]
   \[
   \text{VXCVI (if Invalid Integer Convert)}
   \]

2. \(\text{FR}, \text{FI} \), and \(\text{FPRAF}\) are unchanged.

3. \(\text{FPCC}\) is set to reflect unordered.
For the VSX Vector Compare Single-Precision instructions, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x0000_0000 is placed into its respective word element of VSR[XT].

3. FR, FI, and FPRF are not modified.

For the Vector Double-Precision Compare instructions, do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x0000_0000_0000_0000 is placed into its respective doubleword element of VSR[XT].

3. FR, FI, and FPRF are not modified.

For VSX Vector Convert with round Single-Precision to Half-Precision format (xvcvsphp), do the following.

1. VXSNAN is set to 1.

2. The half-precision representation of a Quiet NaN is placed into the rightmost halfword of its respective word element of VSR[XT]. The contents of the leftmost halfword of its respective word element of VSR[XT] are set to 0.

3. FR, FI, and FPRF are not modified.

For VSX Vector Convert Half-Precision to Single-Precision format (xvcvhpsp), do the following.

1. VXSNAN is set to 1.

2. The half-precision representation of a Quiet NaN is placed into the rightmost halfword of its respective word element of VSR[XT]. The contents of the leftmost halfword of its respective word element of VSR[XT] are set to 0.

3. FR, FI, and FPRF are not modified.
7.4.2 Floating-Point Zero Divide Exception

7.4.2.1 Definition

A Zero Divide exception occurs when a VSX Floating-Point Divide instruction is executed with a zero divisor value and a finite nonzero dividend value.

A Zero Divide exception also occurs when a VSX Floating-Point Reciprocal Estimate instruction or a VSX Floating-Point Reciprocal Square Root Estimate instruction is executed with an operand value of zero.

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

7.4.2.2 Action for ZE=1

When Zero Divide exception is enabled (ZE=1) and a Zero Divide exception occurs, the following actions are taken:

For any of the following instructions,

- VSX Scalar Divide Double-Precision \((xsdivdp)\)
- VSX Scalar Divide Single-Precision \((xsdivsp)\)
- VSX Scalar Divide Quad-Precision \((xsdivqp)\)
- VSX Scalar Reciprocal Estimate Double-Precision \((xsredp)\)
- VSX Scalar Reciprocal Estimate Single-Precision \((xsresp)\)
- VSX Scalar Reciprocal Square Root Estimate Double-Precision \((xsrsqtedp)\)
- VSX Scalar Reciprocal Square Root Estimate Single-Precision \((xsrsqrtesp)\)

do the following.

1. \(ZX\) is set to 1.
2. \(VSR[XT]\) is not modified.
3. \(FR\) and \(FI\) are set to 0. \(FPRF\) is unchanged.

For any of the following instructions,

- VSX Vector Divide Double-Precision \((xvdivdp)\)
- VSX Vector Divide Single-Precision \((xvdivsp)\)
- VSX Vector Reciprocal Estimate Double-Precision \((xvredp)\)
- VSX Vector Reciprocal Estimate Single-Precision \((xvresp)\)
- VSX Vector Reciprocal Square Root Estimate Double-Precision \((xvrsqtedp)\)
- VSX Vector Reciprocal Square Root Estimate Single-Precision \((xvrsqrtesp)\)

do the following.

1. \(ZX\) is set to 1.
2. \(VSR[XT]\) is not modified.
3. \(FR\) and \(FI\) are unchanged. \(FPRF\) is unchanged.
7.4.2.3 Action for ZE=0

When Zero Divide exception is disabled (ZE=0) and a Zero Divide exception occurs, the following actions are taken:

For any of the following instructions,

VSX Scalar Divide Double-Precision (xsdivdp)  
VSX Scalar Divide Single-Precision (xsdivsp)

do the following.

1. \(Z_X\) is set to 1.

2. An Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into doubleword element 0 of \(VSR[XT]\) in double-precision format. The contents of doubleword element 1 of \(VSR[XT]\) are set to 0.

3. \(FR\) and \(FI\) are set to 0. \(FPRF\) is set to indicate the class and sign of the result (± Infinity).

For VSX Scalar Divide Quad-Precision (xsdivqdp), do the following.

1. \(Z_X\) is set to 1.

2. An Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into \(VSR[VRT+32]\) in quad-precision format.

3. \(FR\) and \(FI\) are set to 0. \(FPRF\) is set to indicate the class and sign of the result (± Infinity).

For VSX Vector Divide Double-Precision (xdivdp), do the following.

1. \(Z_X\) is set to 1.

2. For each vector element causing a Zero Divide exception, an Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into its respective doubleword element of \(VSR[XT]\) in double-precision format.

3. \(FR, FI\), and \(FPRF\) are not modified.

For VSX Vector Divide Single-Precision (xdivsp), do the following.

1. \(Z_X\) is set to 1.

2. For each vector element causing a Zero Divide exception, an Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into its respective word element of \(VSR[XT]\) in single-precision format.

3. \(FR, FI\), and \(FPRF\) are not modified.
For any of the following instructions,

VSX Scalar Reciprocal Estimate Double-Precision (xsredp)
VSX Scalar Reciprocal Estimate Single-Precision (xsresp)

VSX Scalar Reciprocal Square Root Estimate Double-Precision (xsrsqrtedp)
VSX Scalar Reciprocal Square Root Estimate Single-Precision (xsrsqrtesp)

do the following.

1. \( ZX \) is set to 1.

2. An Infinity, having the sign of the source operand, is placed into doubleword element 0 of \( VSR[XT] \) in double-precision format. The contents of doubleword element 1 of \( VSR[XT] \) are set to 0.

3. \( FR \) and \( FI \) are set to 0. \( FPRF \) is set to indicate the class and sign of the result (± Infinity).

For any of the following instructions,

VSX Vector Reciprocal Estimate Double-Precision (xsredp)
VSX Vector Reciprocal Square Root Estimate Double-Precision (xsrsqrtedp)

do the following.

1. \( ZX \) is set to 1.

2. For each vector element causing a Zero Divide exception, an Infinity, having the sign of the source operand, is placed into its respective doubleword element of \( VSR[XT] \) in double-precision format.

3. \( FR \), \( FI \), and \( FPRF \) are not modified.

For any of the following instructions,

VSX Vector Reciprocal Estimate Single-Precision (xsresp)
VSX Vector Reciprocal Square Root Estimate Single-Precision (xsrsqrtesp)

do the following.

1. \( ZX \) is set to 1.

2. For each vector element causing a Zero Divide exception, an Infinity, having the sign of the source operand, is placed into its respective word element of \( VSR[XT] \) in single-precision format.

3. \( FR \), \( FI \), and \( FPRF \) are not modified.
7.4.3 Floating-Point Overflow Exception

7.4.3.1 Definition

An Overflow exception occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

7.4.3.2 Action for OE=1

When Overflow exception is enabled (OE=1) and an Overflow exception occurs, the following actions are taken:

For the VSX Vector round and Convert Double-Precision to Single-Precision format (xscvdpsp) instruction:

1. OX is set to 1.
2. If the unbiased exponent of the normalized intermediate result is less than or equal to 318 (Emax+192), the exponent is adjusted by subtracting 192. Otherwise the result is undefined.
3. The adjusted rounded result is placed into word elements 0 and 1 of VSR[XT] in single-precision format. The contents of word elements 2 and 3 of VSR[XT] are set to 0.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For any of the following instructions,

VSX Scalar Double-Precision Arithmetic instructions
xsadddp, xsdivdp, xsmuldp, xssubdp
xsmaddadp, xsmaddmdp, xsmsubadp, xsmsubmdp
xsnmaddadp, xsnmaddmdp, xsnmsubadp, xsnmsubmdp

VSX Vector Reciprocal Estimate Double-Precision (xsredp)

do the following.

1. OX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by subtracting 1536.
3. The adjusted rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are set to 0.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).
For any of the following instructions,

**VSX Scalar Single-Precision Arithmetic instructions**
- xsaddsp, xsdivsp, xsmulsp, xssubsp
- xsmaddasp, xsmaddmsp, xsmsubasp, xsmsubmsp
- xsnmaddasp, xsnmaddmsp, xsnmsubasp, xsnmsubmsp

**VSX Scalar Reciprocal Estimate Single-Precision (xsresp)**

**VSX Vector Reciprocal Square Root Estimate Single-Precision (xrsqrtresp)**

**VSX Scalar Round to Single-Precision (xrsrp)**

do the following.

1. OX is set to 1.
2. The exponent is adjusted by subtracting 192.
3. The adjusted and rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are set to 0.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

For any of the following instructions,

**VSX Scalar Quad-Precision Arithmetic instructions**
- xsaddqp[0], xsdivqp[0], xsmulqp[0], xssqrtqp[0], xssubqp[0]
- xsmaddqp[0], xsmsubqp[0], xsnmaddqp[0], xsnmsubqp[0]

**VSX Scalar Round Quad-Precision to Double-Extended-Precision (xsrqpxp)**

do the following.

1. OX is set to 1.
2. The exponent is adjusted by subtracting 24576.
3. The adjusted, rounded result is placed into VSR[VRT+32] in quad-precision format.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For **VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] (xscvqpdsp)**, do the following.

1. OX is set to 1.
2. The exponent is adjusted by subtracting 1536. If the adjusted exponent is greater than +1023 (Emx), the result is undefined.
3. The adjusted, rounded result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.
   
   0x0000.0000_0000.0000 is placed into doubleword element 1 of VSR[VRT+32].
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).
For VSX Scalar Convert with round Double-Precision to Half-Precision format (**xscvdphp**), do the following.

1. **OX** is set to 1.
2. The exponent is adjusted by subtracting 24. If the adjusted exponent is greater than +15 (\(E_{\max}\)), the result is undefined.
3. The adjusted, rounded result is placed into rightmost halfword of doubleword element 0 of **VSR[XT]** in half-precision format.
   - The contents of the leftmost 3 halfwords of doubleword element 0 of **VSR[XT]** are set to 0.
   - The contents of doubleword element 1 of **VSR[XT]** are set to 0.
4. Unless the result is undefined, **FPRF** is set to indicate the class and sign of the result (±Normal Number).

For any of the following instructions,

- **VSX Vector Double-Precision Arithmetic instructions**
  - **xvadddp, xvdivdp, xvmuldp, xvredp, xvsubdp,**
  - **xvmaddadp, xsmaddmdp, xvmsubadp, xvmsubmdp,**
  - **xvnaddadp, xvnaddmdp, xvnmsubadp, xvnmsubmdp**

- **VSX Vector Single-Precision Arithmetic instructions**
  - **xvaddsp, xvdivsp, xvmulsp, xvresp, xvsubsp**
  - **xvmaddasp, xvmaddmsp, xvmsubasp, xvmsubmsp**
  - **xvnaddasp, xvnaddmsp, xvnmsubasp, xvnmsubmsp**

- **VSX Vector round and Convert Double-Precision to Single-Precision format (**xvcvdpsp**)**

  do the following.

1. **OX** is set to 1.
2. **VSR[XT]** is not modified.
3. **FR, FI**, and **FPRF** are not modified.

For VSX Vector Convert with round Single-Precision to Half-Precision format (**xvcvsphp**), do the following.

1. **OX** is set to 1.
2. **VSR[XT]** is not modified.
3. **FR, FI**, and **FPRF** are not modified.
7.4.3.3 Action for OE=0

When Overflow exception is disabled (OE=0) and an Overflow exception occurs, the following actions are taken:

1. OX and XX are set to 1.
2. The result is determined by the rounding mode (RN) and the sign of the intermediate result as follows:

   **Round to Nearest Even**
   For negative overflow, the result is -Infinity.
   For positive overflow, the result is +Infinity.

   **Round toward Zero**
   For negative overflow, the result is the format’s most negative finite number.
   For positive overflow, the result is the format’s most positive finite number.

   **Round toward +Infinity**
   For negative overflow, the result is the format’s most negative finite number.
   For positive overflow, the result is +Infinity.

   **Round toward -Infinity**
   For negative overflow, the result is -Infinity.
   For positive overflow, the result is the format’s most positive finite number.

For VSX Scalar round and Convert Double-Precision to Single-Precision format (xscvdpsp):

3. The result is placed into word elements 0 and 1 of VSR[XT] as a single-precision value. The contents of word elements 2 and 3 of VSR[XT] are set to 0.
4. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For any of the following instructions,

**VSX Scalar Double-Precision Arithmetic instructions**
xsaddp, xsdivp, xsmulp, xsredp, xssubdp
xsmaddadp, xsmaddmdp, xsmsubadp, xsmsubmdp
xsnmaddadp, xsnmaddmdp, xsnmsubadp, xsnmsubmdp

**VSX Scalar Single-Precision Arithmetic instructions**
xsaddsp, xsdivsp, xsmulsp, xssresp, xssubsp
xsmaddasp, xsmaddmsp, xsmsubasp, xsmsubmsp
xsnmaddasp, xsnmaddmsp, xsnmsubasp, xsnmsubmsp

do the following.

3. The result is placed into doubleword element 0 of VSR[XT] as a double-precision value. The contents of doubleword element 1 of VSR[XT] are set to 0.
4. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.
For any of the following instructions,

**VSX Scalar Quad-Precision Arithmetic instructions**
xsaddqp[0], xsdivqp[0], xsmulqp[0], xssubqp[0]
xsmaddqp[0], xsmsubqp[0], xsnaddqp[0], xsnsubqp[0]

**VSX Scalar Quad-Precision Round to Double-Extended-Precision (xsrpqxp)**
do the following.

3. The result is placed into VSR[VRT+32] in quad-precision format.

4. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For **VSX Scalar Convert with round Quad-Precision to Double-Precision format (xscvqdp)**, do the following.

3. The result is placed into doubleword element 0 of VSR[VRT+32] as a double-precision value.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].

4. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For **VSX Scalar Convert with round Double-Precision to Half-Precision format (xscvdphp)**, do the following.

3. The result is placed into the rightmost halfword of doubleword element 0 of VSR[XT] as a half-precision value.

   The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.

   The contents of doubleword element 1 of VSR[XT] are set to 0.

4. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For any of the following instructions,

For **VSX Vector Double-Precision Arithmetic instructions**
xvadddp, xvdivdp, xvmuldp, xvredp, xvsbldp
xvmaddadp, xvmaddmdp, xvmsubadp, xvmsubmdp
xvnaddadp, xvnaddmdp, xvnsubadp, xvnsubmdp
do the following.

3. For each vector element causing an Overflow exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.

4. FR, FI, and FPRF are not modified.
For any of the following instructions,

VSX Vector Single-Precision Arithmetic instructions
\[ xvaddsp, xvdivsp, xvmuls, xvresp, xvsbsp \]
\[ xvmaddsp, xvmaddmsp, xvmsubasp, xvmsubmsp \]
\[ xvnmadsp, xvnmadmsp, xvnmsubasp, xvnmsubmsp \]

VSX Vector round and Convert Double-Precision to Single-Precision format \((xvcvdpsp)\)
do the following.

3. For each vector element causing an Overflow exception, the result is placed into its respective word element of \(VSR[XT]\) in single-precision format, and for \(xvcvdpsp\), is also placed into bits 32:63 of its respective doubleword element of \(VSX[XT]\).

4. \(FR, FI, \) and \(FPRF\) are not modified.

For VSX Vector Convert with round Single-Precision to Half-Precision format \((xvcvsphp)\), do the following.

3. For each vector element causing an Overflow exception, the result is placed into the rightmost halfword of its respective word element of \(VSR[XT]\) in half-precision format.

The contents of the leftmost halfword of its respective word element of \(VSR[XT]\) are set to 0.

4. \(FR, FI, \) and \(FPRF\) are not modified.
7.4.4 Floating-Point Underflow Exception

7.4.4.1 Definition

Underflow exception is defined separately for the enabled and disabled states:

**Enabled:**
Underflow occurs when the intermediate result is “Tiny”.

**Disabled:**
Underflow occurs when the intermediate result is “Tiny” and there is “Loss of Accuracy”.

A tiny result is detected before rounding, when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is tiny and Underflow exception is disabled (\( \text{UE}=0 \)), the intermediate result is denormalized (see Section 7.3.2.4, “Normalization and Denormalization” on page 513) and rounded (see Section 7.3.2.6, “Rounding” on page 518) before being placed into the target VSR.

*Loss of accuracy* is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

7.4.4.2 Action for UE=1

When Underflow exception is enabled (\( \text{UE}=1 \)) and an Underflow exception occurs, the following actions are taken:

For VSX Scalar round and Convert Double-Precision to Single-Precision format (\( \text{xscvdpsp} \)), do the following.

1. \( \text{UX} \) is set to 1.

2. The exponent of the normalized intermediate result is adjusted by adding 192. If the adjusted unbiased exponent is less than \(-126\) (\( \text{E}_{\text{min}} \)), the result is undefined.

3. The adjusted rounded result is placed into word elements 0 and 1 of \( \text{VSR}[\text{XT}] \) in single-precision format. The contents of word elements 2 and 3 of \( \text{VSR}[\text{XT}] \) are undefined.

4. Unless the result is undefined, \( \text{FPRF} \) is set to indicate the class and sign of the result (\( \pm\text{Normal Number} \)).
For any of the following instructions,

**VSX Scalar Double-Precision Arithmetic instructions**

- xsadddp, xsdivdp, xsmuldp, xssubdp
- xsaddadp, xsaddmdp, xsmaddadp, xssubadp
- xsmaddadp, xsmsubadp, xsmsubmdp

**VSX Scalar Double-Precision Reciprocal Estimate (xsredp)**

do the following.

1. UX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by adding 1536.
3. The adjusted rounded result is placed into word elements 0 and 1 of VSR[XT] in single-precision format. The contents of word elements 2 and 3 of VSR[XT] are set to 0
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

For any of the following instructions,

**VSX Scalar Quad-Precision Arithmetic instructions**

- xsaddqp, xsdivqp, xsmulqp, xssubqp
- xsaddadp, xsaddmdp, xsmaddadp, xssubadp
- xsmaddadp, xsmsubadp, xsmsubmdp

**VSX Scalar Round Quad-Precision to Double-Extended-Precision (xsrqpxp)**

do the following.

1. UX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by adding 24576.
3. The adjusted, rounded result is placed into VSR[VRT+32] in quad-precision format.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

For **VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd]** (xscvqdp[O]), do the following.

1. UX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by adding 1536. If the adjusted unbiased exponent is less than -1022 (Emi n), the result is undefined.
3. The adjusted, rounded result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.

4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).
For any of the following instructions,

**VSX Scalar Single-Precision Arithmetic instructions**
- `xsaddsp`, `xsdivsp`, `xsmulsp`, `xssubsp`
- `xsmaddasp`, `xsmaddmsp`, `xsmsubasp`, `xsmsubmsp`
- `xsnmaddasp`, `xsnmaddmsp`, `xsnmsubasp`, `xsnmsubmsp`

**VSX Scalar Single-Precision Reciprocal Estimate (xsresp)**

do the following.

1. `UX` is set to 1.
2. The exponent of the normalized intermediate result is adjusted by adding 192. If the adjusted unbiased exponent is less than \(-126\) (\(E_{\text{min}}\)), the result is undefined.
3. The adjusted rounded result is placed into doubleword element 0 of \(\text{VSR}[\text{XT}]\) in double-precision format. The contents of doubleword element 1 of \(\text{VSR}[\text{XT}]\) are set to 0.
4. Unless the result is undefined, \(\text{FPRF}\) is set to indicate the class and sign of the result (\(\pm \text{Normal Number}\)).

---

**Programming Note**

The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an Underflow exception, to simulate a “trap disabled” environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized and correctly rounded.

For **VSX Scalar Convert with round Double-Precision to Half-Precision with round (xscvdphp)**, do the following.

1. `UX` is set to 1.
2. The exponent of the normalized intermediate result is adjusted by adding 24. If the adjusted unbiased exponent is less than \(-14\), the result is undefined.
3. The adjusted, rounded result is placed into rightmost halfword of doubleword element 0 of \(\text{VSR}[\text{XT}]\) in half-precision format.

   The contents of the leftmost 3 halfwords of doubleword element 0 of \(\text{VSR}[\text{XT}]\) are set to 0.

   The contents of doubleword element 1 of \(\text{VSR}[\text{XT}]\) are set to 0.
4. Unless the result is undefined, \(\text{FPRF}\) is set to indicate the class and sign of the result (\(\pm \text{Normal Number}\)).
For any of the following instructions,

**VSX Vector Double-Precision Arithmetic instructions**
- `xvadddp`, `xvdvdp`, `xvmuldp`, `xvsubdp`
- `xvmaddadp`, `xvmaddmdp`, `xvmsubadp`, `xvmsubmdp`
- `xvnaddadp`, `xvnaddmdp`, `xvnmsubadp`, `xvnmsubmdp`

**VSX Vector Single-Precision Arithmetic instructions**
- `xvadsp`, `xvdvsp`, `xvmulp`, `xvsbisp`
- `xvmaddasp`, `xvmaddmsp`, `xvmsubasp`, `xvmsubmsp`
- `xvnmaddasp`, `xvnaddmsp`, `xvnmsubasp`, `xvnmsubmsp`

**VSX Vector Reciprocal Estimate Double-Precision** (`xvredp`)
**VSX Vector Reciprocal Estimate Single-Precision** (`xvresp`)

**VSX Vector round and Convert Double-Precision to Single-Precision format** (`xvcvdpsp`)
**VSX Vector Convert with round Single-Precision to Half-Precision format** (`xvcvspshp`)

do the following.

1. `UX` is set to 1.
2. `VSR[XT]` is not modified.
3. `FR`, `FI`, and `FPRF` are not modified.

### 7.4.4.3 Action for UE=0

When Underflow exception is disabled (`UE=0`) and an Underflow exception occurs, the following actions are taken:

For **VSX Scalar round and Convert Double-Precision to Single-Precision format** (`xsvcdpshp`), do the following.

1. `UX` is set to 1.
2. The result is placed into word elements 0 and 1 of `VSR[XT]` as a single-precision value. The contents of word elements 2 and 3 of `VSR[XT]` are set to 0.
3. `FPRF` is set to indicate the class and sign of the result.

For any of the following instructions,

**VSX Scalar Double-Precision Arithmetic instructions**
- `xsadddp`, `xsdvdp`, `xsmuldp`, `xssubdp`
- `xsmsaddadp`, `xsmsaddmdp`, `xsmssubadp`, `xsmsubmdp`
- `xsmsnaddadp`, `xsmsnaddmdp`, `xsmsnmsubadp`, `xsmsnmsubmdp`

**VSX Scalar Single-Precision Arithmetic instructions**
- `xsaddsp`, `xsdvsp`, `xsmulsp`, `xssubsp`
- `xsmsaddasp`, `xsmsaddmsp`, `xsmssubasp`, `xsmsubmsp`
- `xsmsnaddasp`, `xsmsnaddmsp`, `xsmsnmsubasp`, `xsmsnmsubmsp`

**VSX Scalar Reciprocal Estimate Double-Precision** (`xsredp`)
**VSX Scalar Reciprocal Estimate Single-Precision** (`xsresp`)
do the following.

1. UX is set to 1.

2. The result is placed into doubleword element 0 of \(VSR[XT]\) in double-precision format. The contents of doubleword element 1 of \(VSR[XT]\) are set to 0.

3. FPRF is set to indicate the class and sign of the result.

For any of the following instructions,

\[
\text{VSX Scalar Quad-Precision Arithmetic instructions} \\
\text{xsaddqp}[o], \text{xsdvigqp}[o], \text{xsmulqp}[o], \text{xssubqp}[o] \\
\text{xsmaddqp}[o], \text{xsmsubqp}[o], \text{xsnmaddqp}[o], \text{xsnmsubqp}[o]
\]

\[
\text{VSX Scalar Round Quad-Precision to Double-Extended-Precision (xsrqpxp)}
\]

do the following.

1. UX is set to 1.

2. The result is placed into \(VSR[VRT+32]\) in quad-precision format.

3. FPRF is set to indicate the class and sign of the result.

For \text{VSX Scalar Convert with round Quad-Precision to Double-Precision format (xscvqdp)}, do the following.

1. UX is set to 1.

2. The result is placed into doubleword element 0 of \(VSR[VRT+32]\) in double-precision format.

\[
0x0000_0000_0000_0000 
\]

is placed into doubleword element 1 of \(VSR[VRT+32]\).

3. FPRF is set to indicate the class and sign of the result.

For \text{VSX Scalar Convert with round Double-Precision to Half-Precision format (xscvdphp)}, do the following.

1. UX is set to 1.

2. The result is placed into the rightmost halfword of doubleword element 0 of \(VSR[XT]\) as a half-precision value.

   The contents of the leftmost 3 halfwords of doubleword element 0 of \(VSR[XT]\) are set to 0.

   The contents of doubleword element 1 of \(VSR[XT]\) are set to 0.

3. FPRF is set to indicate the class and sign of the result.

For any of the following instructions,

\[
\text{VSX Vector Double-Precision Arithmetic instructions} \\
xvadddp, xvdivdp, xvmuldp, xsvsubdp \\
xvmaddadp, xvmaddmdp, xvmsubadp, xvmsubmdp \\
xvnmaddadp, xvnmaddmdp, xvnmsubadp, xvnmsubmdp
\]

\[
\text{VSX Vector Reciprocal Estimate Double-Precision (xvredp)}
\]
do the following.

1. UX is set to 1.

2. For each vector element causing an Underflow exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.

3. FR, FI, and FPRF are not modified.

For any of the following instructions,

- **VSX Vector Single-Precision Arithmetic instructions**
  - xvaddsp, xdivsp, xvmulp, xvsbsp
  - xvmaddsp, xvmaddmsp, xvmsubasp, xvmsubmsp
  - xvnaddasp, xvnaddmsp, xvnsubasp, xvnsubmsp

- **VSX Vector Reciprocal Estimate Single-Precision (xvresp)**

- **VSX Vector round and Convert Double-Precision to Single-Precision format (xvcvdpsp)**

  do the following.

  1. UX is set to 1.

  2. For each vector element causing an Underflow exception, the result is placed into its respective word element of VSR[XT] in single-precision format, and for xvcvdpsp, is also placed into bits 32:63 of its respective doubleword element of VSR[XT].

  3. FR, FI, and FPRF are not modified.

For **VSX Vector Convert with round Single-Precision to Half-Precision format (xvcvshp)**, do the following.

  1. UX is set to 1.

  2. For each vector element causing an Underflow exception, the result is placed into the rightmost halfword of its respective word element of VSR[XT] in half-precision format.

     The contents of the leftmost halfword of its respective word element of VSR[XT] are set to 0.

  3. FR, FI, and FPRF are not modified.
7.4.5 Floating-Point Inexact Exception

7.4.5.1 Definition

An Inexact exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled Overflow exception or an enabled Underflow exception, an Inexact exception also occurs only if the significands of the rounded result and the intermediate result differ.)

2. The rounded result overflows and Overflow exception is disabled.

The action to be taken depends on the setting of the Inexact Exception Enable bit of the FPSCR.

7.4.5.2 Action for XE=1

When Inexact exception is enabled (\( UE=1 \)) and an Inexact exception occurs, the following actions are taken:

For the \textit{VSX Scalar round and Convert Double-Precision to Single-Precision format} (\texttt{xscvdpsp}) instruction, do the following.

1. \texttt{XX} is set to 1.
2. The result is placed into word elements 0 and 1 of \texttt{VSR[XT]} in single-precision format. The contents of word elements 2-3 of \texttt{VSR[XT]} are set to 0.
3. \texttt{FPRF} is set to indicate the class and sign of the result.

For any of the following instructions,

- \texttt{xsadddp, xssubdp, xsmuldp, xsdivdp, xsqrtdp}
- \texttt{xsmaddadp, xsmaddmdp, xmssubadp, xmssubmdp}
- \texttt{xsnmaddadp, xsnmaddmdp, xsnmsubadp, xsnmsubmdp}

- \texttt{xssaddsp, xssubsp, xsmulsp, xsdivsp, xsqrtspsp}
- \texttt{xsmaddasp, xsmaddmsp, xmssubasp, xmssubmsp}
- \texttt{xsnmaddasp, xsnmaddmsp, xsnmsubasp, xsnmsubmsp}

- \texttt{xsrsp, xsrspq, xsrspsp, xsrspqsp}

- \texttt{xsrsp, xsrspq, xsrspsp, xsrspqsp}

- \texttt{xsrsp, xsrspq, xsrspsp, xsrspqsp}

Program Notes:

- In some implementations, enabling Inexact exceptions can degrade performance more than does enabling other types of floating-point exception.
do the following.

1. \texttt{XX} is set to 1.

2. The result is placed into doubleword element 0 of \texttt{VSR[XT]} in double-precision format. The contents of doubleword element 1 of \texttt{VSR[XT]} are set to 0.

3. \texttt{FPRF} is set to indicate the class and sign of the result.

For any of the following instructions,

\texttt{VSX Scalar Convert with round to zero Double-Precision to Signed Word format (xscvdpsxws)}
\texttt{VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format (xscvdpuws)}

do the following.

1. \texttt{XX} is set to 1.

2. The result is placed into word element 1 of \texttt{VSR[XT]}. The contents of word elements 0, 2, and 3 of \texttt{VSR[XT]} are set to 0.

3. \texttt{FPRF} is set to indicate the class and sign of the result.

For any of the following instructions,

\texttt{VSX Scalar Quad-Precision Arithmetic instructions}
\texttt{xsaddqp[o], xsdivqp[o], xsmulqp[o], xsqrtqp[o], xssubqp[o]}
\texttt{xsmaddqp[o], xsmsubqp[o], xsnmaddqp[o], xsnmsubqp[o]}

\texttt{VSX Scalar Round to Quad-Precision Integer with Inexact (xsrqpix)}
\texttt{VSX Scalar Round Quad-Precision to Double-Extended Precision (xsrqpxp)}

do the following.

1. \texttt{XX} is set to 1.

2. The result is placed into \texttt{VSR[VRT+32]} in quad-precision format.

3. \texttt{FR} is set to indicate if the rounded result was incremented. \texttt{FI} is set to 1. \texttt{FPRF} is set to indicate the class and sign of the result.

For \texttt{VSX Scalar Convert with round Quad-Precision to Double-Precision format (xscvqdpd)}, do the following.

1. \texttt{XX} is set to 1.

2. The result is placed into doubleword element 0 of \texttt{VSR[VRT+32]} in double-precision format.

\hspace{1cm} \texttt{0x0000_0000_0000_0000} is placed into doubleword element 1 of \texttt{VSR[VRT+32]}.

3. \texttt{FR} is set to indicate if the rounded result was incremented. \texttt{FI} is set to 1. \texttt{FPRF} is set to indicate the class and sign of the result.
For **VSX Scalar truncate & Convert Quad-Precision to Signed Doubleword** (*xscvqpsdz*), do the following.

1. \( XX \) is set to 1.
2. The result is placed into doubleword element 0 of VSR[XT] in signed integer format.
   \( 0x0000_0000_0000_0000 \) is placed into doubleword element 1 of VSR[VRT+32].
3. \( FR \) is set to 0. \( FI \) is set to 1. \( FPRF \) is undefined.

For **VSX Scalar truncate & Convert Quad-Precision to Signed Word** (*xscvqpswz*), do the following.

1. \( XX \) is set to 1.
2. The result is placed into word element 1 of VSR[XT] in signed integer format.
   \( 0x0000_0000 \) is placed into word elements 0, 2, and 3 of VSR[VRT+32].
3. \( FR \) is set to 0. \( FI \) is set to 1. \( FPRF \) is undefined.

For **VSX Scalar truncate & Convert Quad-Precision to Unsigned Doubleword** (*xscvqpwdz*), do the following.

1. \( XX \) is set to 1.
2. The result is placed into doubleword element 0 of VSR[XT] in unsigned integer format.
   \( 0x0000_0000_0000_0000 \) is placed into doubleword element 1 of VSR[VRT+32].
3. \( FR \) is set to 0. \( FI \) is set to 1. \( FPRF \) is undefined.

For **VSX Scalar truncate & Convert Quad-Precision to Unsigned Word** (*xscvqpwz*), do the following.

1. \( XX \) is set to 1.
2. The result is placed into word element 1 of VSR[XT] in unsigned integer format.
   \( 0x0000_0000 \) is placed into word elements 0, 2, and 3 of VSR[VRT+32].
3. \( FR \) is set to 0. \( FI \) is set to 1. \( FPRF \) is undefined.

For **VSX Scalar Convert with round Double-Precision to Half-Precision truncate** (*xscvdphp*), do the following.

1. \( XX \) is set to 1.
2. The result is placed into the rightmost halfword of doubleword element 0 of VSR[XT] as a half-precision value.
   The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.
   The contents of doubleword element 1 of VSR[XT] are set to 0.
3. \( FR \) is set to indicate if the rounded result was incremented. \( FI \) is set to 1. \( FPRF \) is set to indicate the class and sign of the result.
For any of the following instructions,

**VSX Scalar Double-Precision Arithmetic instructions**
- `xsadddp`, `xssubdp`, `xsmuldp`, `xsdvdp`, `xssqrtdp`
- `xsmaddadp`, `xsmaddmdp`, `xsmsubadp`, `xsmsubmdp`
- `xsnmaddadp`, `xsnmaddmdp`, `xsnmsubadp`, `xsnmsubmdp`

**VSX Scalar Single-Precision Arithmetic instructions**
- `xsaddsp`, `xssubsp`, `xsmulsp`, `xsdvsp`, `xssqrtsp`
- `xsmaddasp`, `xsmaddmsp`, `xsmsubasp`, `xsmsubmsp`
- `xsnmaddasp`, `xsnmaddmsp`, `xsnmsubasp`, `xsnmsubmsp`

**VSX Scalar Reciprocal Estimate instructions**
- `xsredp`, `xsrqtedp`, `xsresp`, `xsrqresp`

**VSX Scalar Round to Single-Precision (xsrsp)**

**VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode (xsrdpic)**

**VSX Scalar Convert with round Signed Doubleword to Double-Precision format (xscvsxddp)**

**VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format (xscvuxddp)**

**VSX Scalar Convert with round Signed Doubleword to Single-Precision format (xscvsxdsp)**

**VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format (xscvuxdsp)**

For the following:

1. `XX` is set to 1.
2. `VSR[XT]` is not modified.
3. `FR`, `FI`, and `FPRF` are not modified.

For the VSX Vector Convert with round Single-Precision to Half-Precision format (xvcvsphp) instruction, do the following.

1. `XX` is set to 1.
2. `VSR[XT]` is not modified.
3. `FR`, `FI`, and `FPRF` are not modified.
7.4.5.3 Action for XE=0

When Inexact exception is disabled (XE=0) and an Inexact exception occurs, the following actions are taken:

For VSX Scalar round and Convert Double-Precision to Single-Precision format (xscvdpsp), do the following.

1. \(XX\) is set to 1.
2. The result is placed into word elements 0 and 1 of \(VSR[XT]\) as a single-precision value. The contents of word elements 2-3 of \(VSR[XT]\) are set to 0.
3. \(FPRF\) is set to indicate the class and sign of the result.

For any of the following instructions,

<table>
<thead>
<tr>
<th>VSX Scalar Double-Precision Arithmetic instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsadddp, xssubdp, xsmuldp, xsdivdp, xssqrtdp</td>
</tr>
<tr>
<td>xsmaddadp, xsmaddmdp, xsmsubadp, xsmsubmdp</td>
</tr>
<tr>
<td>xsnmaddadp, xsnmaddmdp, xsnmsubadp, xsnmsubmdp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VSX Scalar Single-Precision Arithmetic instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsaddsp, xssubsp, xsmulsp, xsdivsp, xssqrtsp</td>
</tr>
<tr>
<td>xsmaddasp, xsmaddmsp, xsmsubasp, xsmsubmsp</td>
</tr>
<tr>
<td>xsnmaddasp, xsnmaddmsp, xsnmsubasp, xsnmsubmsp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VSX Scalar Round to Single-Precision (xsrsp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode (xsrdpic)</td>
</tr>
<tr>
<td>VSX Scalar Convert with round Signed Doubleword to Double-Precision format (xscvsxddp)</td>
</tr>
<tr>
<td>VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format (xscvuxddp)</td>
</tr>
</tbody>
</table>

do the following.

1. \(XX\) is set to 1.
2. The result is placed into doubleword element 0 of \(VSR[XT]\) as a double-precision value. The contents of doubleword element 1 of \(VSR[XT]\) are set to 0.
3. \(FPRF\) is set to indicate the class and sign of the result.

For any of the following instructions,

<table>
<thead>
<tr>
<th>VSX Scalar Convert with round to zero Double-Precision To Signed Word format (xscvdpsxws)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSX Scalar Convert with round to zero Double-Precision To Unsigned Word format (xscvdpuxws)</td>
</tr>
</tbody>
</table>

do the following.

1. \(XX\) is set to 1.
2. The result is placed into word elements 0 and 1 of \(VSR[XT]\). The contents of word elements 2 and 3 of \(VSR[XT]\) are set to 0.
3. \(FPRF\) is set to indicate the class and sign of the result.
For VSX Scalar Convert with round Quad-Precision to Double-Precision format (*xscvqpdp*), do the following.

1. **XX** is set to 1.
2. The result is placed into the rightmost halfword of doubleword element 0 of *VSR[XT]* as a half-precision value.
   - The contents of the leftmost 3 halfwords of doubleword element 0 of *VSR[XT]* are set to 0.
   - The contents of doubleword element 1 of *VSR[XT]* are set to 0.
3. **FR** is set to indicate if the rounded result was incremented. **FI** is set to 1. **FPRF** is set to indicate the class and sign of the result.

For any of the following instructions,

**VSX Vector Double-Precision Arithmetic instructions**

- *xvadddp*
- *xvsubdp*
- *xvmuldp*
- *xvdivdp*
- *xvsqrtdp*
- *xvmaddadp*
- *xvmaddmdp*
- *xvmsubadp*
- *xvmsubmdp*
- *xvnmsaddadp*
- *xvnmsaddmdp*
- *xvnmsubadp*
- *xvnmsubmdp*

do the following.

1. **XX** is set to 1.
2. For each vector element causing an Inexact exception, the result is placed into its respective doubleword element of *VSR[XT]* in double-precision format.
3. **FR**, **FI**, and **FPRF** are not modified.

For any of the following instructions,

**VSX Scalar Quad-Precision Arithmetic instructions**

- *xsaddqp[o]*
- *xsdqvoq[p]*
- *xsmulqp[o]*
- *xssqrtqp[o]*
- *xssubqp[o]*
- *xsmaddqp[o]*
- *xsmsubqp[o]*
- *xsnmaddqp[o]*
- *xsnnsubqp[o]*

**VSX Scalar Round Quad-Precision to Double-Extended-Precision (xsrqpxp)**

**VSX Scalar Round to Quad-Precision Integer with Inexact (xsrqpix)**

do the following.

1. **XX** is set to 1.
2. The result is placed into *VSR[VRT+32]* in quad-precision format.
3. **FR** is set to indicate if the rounded result was incremented. **FI** is set to 1. **FPRF** is set to indicate the class and sign of the result.

For VSX Scalar round & Convert Quad-Precision to Double-Precision (*xscvqpdp*), do the following.

1. **XX** is set to 1.
2. The result is placed into doubleword element 0 of *VSR[VRT+32]* in double-precision format.
   - `0x0000_0000_0000_0000` is placed into doubleword element 1 of *VSR[VRT+32]*.
3. **FR** is set to indicate if the rounded result was incremented. **FI** is set to 1. **FPRF** is set to indicate the class and sign of the result.
For any of the following instructions,

- **VSX Scalar truncate & Convert Quad-Precision to Signed Doubleword** (*xscvqpsdz*)
- **VSX Scalar truncate & Convert Quad-Precision to Signed Word** (*xscvqpswz*)

do the following.

1. \( \text{XX} \) is set to 1.
2. The result is placed into doubleword element 0 of \( VSR[VRT+32] \) in signed integer format.
   
   \( 0x0000_0000_0000_0000 \) is placed into doubleword element 1 of \( VSR[VRT+32] \).
3. \( FR \) is set to 0. \( FI \) is set to 1. \( FPRF \) is undefined.

For any of the following instructions,

- **VSX Scalar truncate & Convert Quad-Precision to Unsigned Doubleword** (*xscvq pudz*)
- **VSX Scalar truncate & Convert Quad-Precision to Unsigned Word** (*xscvqp uzwz*)

do the following.

1. \( \text{XX} \) is set to 1.
2. The result is placed into doubleword element 0 of \( VSR[VRT+32] \) in unsigned integer format.
   
   \( 0x0000_0000_0000_0000 \) is placed into doubleword element 1 of \( VSR[VRT+32] \).
3. \( FR \) is set to 0. \( FI \) is set to 1. \( FPRF \) is undefined.

For **VSX Vector Convert with round Single-Precision to Half-Precision format** (*xvcv spdh*), do the following.

1. \( \text{XX} \) is set to 1.
2. For each vector element causing an Underflow exception, the result is placed into the rightmost halfword of its respective word element of \( VSR[XT] \) in half-precision format.
   
   The contents of the leftmost halfword of its respective word element of \( VSR[XT] \) are set to 0.
3. \( FR, FI, \) and \( FPRF \) are not modified.

For any of the following instructions,

- **VSX Vector Single-Precision Arithmetic instructions**
  - *xvaddsp*, *xvsubsp*, *xvmulp*, *xdivsp*, *xvsqrtsp*
  - *xvmaddasp*, *xvmaddmsp*, *xvmsubasp*, *xvmsubmsp*
  - *xvnmsubasp*, *xvnmsubmsp*, *xvnmsubmsp*

do the following.

1. \( \text{XX} \) is set to 1.
2. For each vector element causing an Inexact exception, the result is placed into its respective word element of \( VSR[XT] \) in single-precision format.
3. \( FR, FI, \) and \( FPRF \) are not modified.
7.5 VSX Storage Access Operations

The VSX Storage Access instructions compute the effective address (EA) of the storage to be accessed as described in Power ISA Book I.

7.5.1 Accessing Aligned Storage Operands

The following quadword-aligned array, AH, consists of 8 halfwords.

```c
short AW[4] = { 0x0001_0203, 0x0405_0607, 0x0809_0A0B, 0x0C0D_0E0F };
```

Figure 121 illustrates the Big-Endian storage image of array AW.

![Big-Endian storage image of array AW](image1)

Figure 121.Big-Endian storage image of array AW

Figure 122 illustrates the Little-Endian storage image of array AW.

![Little-Endian storage image of array AW](image2)

Figure 122.Little-Endian storage image of array AW

Figure 123 shows the result of loading that quadword into a VSR or, equivalently, shows the contents that must be in a VSR if storing that VSR is to produce the storage contents shown in Figure 121 for Big-Endian. Note that Figure shows the effect of loading the quadword from both Big-Endian storage and Little-Endian storage.

![Vector-Scalar Register contents for aligned quadword Load or Store VSX Vector](image3)
7.5.2 Accessing Unaligned Storage Operands

The following array, B, consists of 5 word elements.

```c
int B[5];
B[0] = 0x01234567;
B[1] = 0x00112233;
B[2] = 0x44556677;
B[3] = 0x8899AABB;
B[4] = 0xCCDDEEFF;
```

Figure 124 illustrates both Big-Endian and Little-Endian storage images of array B.

Loading an Unaligned Quadword from Big-Endian Storage

Loading elements from elements 1 through 4 of B (see Figure 124) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Big-Endian byte ordering.

Figure 125. Process to load unaligned quadword from Big-Endian storage using Load VSX Vector Word*4 Indexed

Loading an Unaligned Quadword from Little-Endian Storage

Loading elements from elements 1 through 4 of B (see Figure 124) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Little-Endian byte ordering.

Figure 126. Process to load unaligned quadword from Little-Endian storage Load VSX Vector Word*4 Indexed
Storing an Unaligned Quadword to Big-Endian Storage

Storing a VSR to elements 1 through 4 of B (see Figure 124) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Big-Endian byte ordering.

Big-Endian storage image of array B

0x0000: 01 23 45 67 00 11 22 33 44 55 66 77 88 99 AA BB
       0x0010: CC DD EE FF

Xs: F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA BB FC FD FE FF

# Assumptions
GPR[Ra] = address of B
GPR[Rb] = 4 (index to B[1])

stxvw4x Xs, Ra, Rb

0x0000: 01 23 45 67 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA BB
       0x0010: FC FD FE FF

Figure 127. Process to store unaligned quadword to Big-Endian storage using Store VSX Vector Word*4 Indexed

Storing an Unaligned Quadword to Little-Endian Storage

Storing a VSR to elements 1 through 4 of B (see Figure 124) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Little-Endian byte ordering.

Little-Endian storage image of array B

0x0000: 67 45 23 01 33 22 11 00 77 66 55 44 BB AA 99 88
       0x0010: FF EE DD CC

Xs: F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA BB FC FD FE FF

# Assumptions
GPR[A] = address of B
GPR[B] = 4 (index to B[1])

stxvw4x Xs, Ra, Rb

0x0000: 67 45 23 01 F3 F2 F1 F0 F7 F6 F5 F4 F8 F9 FA FB
       0x0010: FF FE FD FC

Figure 128. Process to store unaligned quadword to Little-Endian storage using Store VSX Vector Word*4 Indexed

7.5.3 Storage Access Exceptions

Storage accesses cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (Store only), or if the program attempts to access storage that is unavailable.
7.6 VSX Instruction Set

7.6.1 VSX Instruction Set Summary

7.6.1.1 VSX Storage Access Instructions

Load VSX Scalar instructions place a copy of the contents of the addressed byte, halfword, word, or doubleword in storage into the left-most doubleword element of the target VSR. For integer byte, halfword, and word forms, the data are placed into the rightmost byte, halfword, or word of the doubleword, and the leftmost bits of the doubleword are set to 0 (or set to the copy of the sign bit for \textit{xsiwax}). For the single-precision floating-point word form, the data is converted to double-precision format and placed into the doubleword. The contents of the right-most doubleword element of the target VSR are set to 0.

Store VSX Scalar instructions place a copy of the contents of the leftmost doubleword element (or portions of) in the source VSR into the addressed byte, halfword, word or doubleword in storage. For integer byte, halfword, and word forms, the rightmost byte, halfword, or word of the doubleword are stored.

Load VSX Vector instructions load a quadword from storage as a vector of 16 byte elements, 8 halfword elements, 4 word elements, 2 doubleword elements or a quadword element into a VSR.

Load VSX Vector & Splat instructions load a word or doubleword from storage and replicate the data into the 4 words or 2 doublewords of a VSR.

Store VSX Vector instructions store a vector of 16 byte elements, 8 halfword elements, 4 word elements, 2 doubleword elements or a quadword element from a VSR into a quadword in storage.

Load VSX Vector with Length instruction loads from 0 to 16 bytes into a VSR.

Store VSX Vector with Length instruction stores from 0 to 16 bytes from a VSR.

Load VSX Vector Paired instructions load an octword (32 bytes) from storage into two sequential VSRs (i.e., a vector of 32 byte elements, 16 halfword elements, 8 word elements, 4 doubleword elements or 2 quadword elements).

VSX Vector Store Paired instructions store the contents of two sequential VSRs into an octword (32 bytes) in storage (i.e., a vector of 32 byte elements, 16 halfword elements, 8 word elements, 4 doubleword elements or 2 quadword elements).

### VSX Scalar Storage Access Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxsd</td>
<td>Load VSX Scalar Doubleword</td>
<td>610</td>
</tr>
<tr>
<td>lxsdx</td>
<td>Load VSX Scalar Doubleword Indexed</td>
<td>611</td>
</tr>
<tr>
<td>lxsiwx</td>
<td>Load VSX Scalar as Integer Word Algebraic Indexed</td>
<td>613</td>
</tr>
<tr>
<td>lxsiwzx</td>
<td>Load VSX Scalar as Integer Word &amp; Zero Indexed</td>
<td>614</td>
</tr>
<tr>
<td>lxssp</td>
<td>Load VSX Scalar Single-Precision</td>
<td>615</td>
</tr>
<tr>
<td>lxsspx</td>
<td>Load VSX Scalar Single-Precision Indexed</td>
<td>616</td>
</tr>
<tr>
<td>plxsd</td>
<td>Prefixed Load VSX Scalar Doubleword</td>
<td>610</td>
</tr>
<tr>
<td>plxssp</td>
<td>Prefixed Load VSX Scalar Single-Precision</td>
<td>615</td>
</tr>
</tbody>
</table>

Table 8. VSX Scalar Load Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pstxsd</td>
<td>Prefixed Store VSX Scalar Doubleword</td>
<td>638</td>
</tr>
<tr>
<td>pstxssp</td>
<td>Prefixed Store VSX Scalar Single-Precision</td>
<td>642</td>
</tr>
<tr>
<td>stxsd</td>
<td>Store VSX Scalar Doubleword</td>
<td>638</td>
</tr>
<tr>
<td>stxsdx</td>
<td>Store VSX Scalar Doubleword Indexed</td>
<td>639</td>
</tr>
<tr>
<td>stxisbx</td>
<td>Store VSX Scalar as Integer Byte Indexed</td>
<td>640</td>
</tr>
</tbody>
</table>

Table 9. VSX Scalar Store Instructions
### 7.6.1.1.2 VSX Vector Storage Access Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stxsihx</td>
<td>Store VSX Scalar as Integer Halfword Indexed</td>
<td>640</td>
</tr>
<tr>
<td>stxsiwx</td>
<td>Store VSX Scalar as Integer Word Indexed</td>
<td>641</td>
</tr>
<tr>
<td>stxsssp</td>
<td>Store VSX Scalar Single-Precision</td>
<td>642</td>
</tr>
<tr>
<td>stxsspx</td>
<td>Store VSX Scalar Single-Precision Indexed</td>
<td>643</td>
</tr>
</tbody>
</table>

Table 9. VSX Scalar Store Instructions

### 7.6.1.2 VSX Vector Storage Access Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxv</td>
<td>Load VSX Vector</td>
<td>617</td>
</tr>
<tr>
<td>lxvb16x</td>
<td>Load VSX Vector Byte*16 Indexed</td>
<td>618</td>
</tr>
<tr>
<td>lxvd2x</td>
<td>Load VSX Vector Doubleword*2 Indexed</td>
<td>619</td>
</tr>
<tr>
<td>lxvh8x</td>
<td>Load VSX Vector Halfword*8 Indexed</td>
<td>634</td>
</tr>
<tr>
<td>lxvw4x</td>
<td>Load VSX Vector Word*4 Indexed</td>
<td>635</td>
</tr>
<tr>
<td>lxvx</td>
<td>Load VSX Vector Indexed</td>
<td>625</td>
</tr>
<tr>
<td>plxv</td>
<td>Prefixed Load VSX Vector</td>
<td>617</td>
</tr>
</tbody>
</table>

Table 10. VSX Vector Load Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxvdlsx</td>
<td>Load VSX Vector Doubleword &amp; Splat Indexed</td>
<td>633</td>
</tr>
<tr>
<td>lxvwxs</td>
<td>Load VSX Vector Word &amp; Splat Indexed</td>
<td>636</td>
</tr>
</tbody>
</table>

Table 11. VSX Vector Load & Splat Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxvrbx</td>
<td>Load VSX Vector Rightmost Byte Indexed</td>
<td>627</td>
</tr>
<tr>
<td>lxvrdx</td>
<td>Load VSX Vector Rightmost Doubleword Indexed</td>
<td>628</td>
</tr>
<tr>
<td>lxvrhx</td>
<td>Load VSX Vector Rightmost Halfword Indexed</td>
<td>629</td>
</tr>
<tr>
<td>lxvrmx</td>
<td>Load VSX Vector Rightmost Word Indexed</td>
<td>630</td>
</tr>
</tbody>
</table>

Table 12. VSX Vector Load Rightmost Element Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxvl</td>
<td>Load VSX Vector with Length</td>
<td>621</td>
</tr>
<tr>
<td>lxvll</td>
<td>Load VSX Vector with Length Left-justified</td>
<td>623</td>
</tr>
</tbody>
</table>

Table 13. VSX Vector Load with Length Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pstxv</td>
<td>Prefixed Store VSX Vector</td>
<td>644</td>
</tr>
<tr>
<td>stxv</td>
<td>Store VSX Vector</td>
<td>644</td>
</tr>
<tr>
<td>stxvb16x</td>
<td>Store VSX Vector Byte*16 Indexed</td>
<td>645</td>
</tr>
<tr>
<td>stxvd2x</td>
<td>Store VSX Vector Doubleword*2 Indexed</td>
<td>646</td>
</tr>
<tr>
<td>stxvh8x</td>
<td>Store VSX Vector Halfword*8 Indexed</td>
<td>647</td>
</tr>
<tr>
<td>stxvw4x</td>
<td>Store VSX Vector Word*4 Indexed</td>
<td>653</td>
</tr>
<tr>
<td>stxvx</td>
<td>Store VSX Vector Indexed</td>
<td>656</td>
</tr>
</tbody>
</table>

Table 14. VSX Vector Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stxvrbx</td>
<td>Store VSX Vector Rightmost Byte Indexed</td>
<td>651</td>
</tr>
<tr>
<td>stxvrdx</td>
<td>Store VSX Vector Rightmost Doubleword Indexed</td>
<td>651</td>
</tr>
<tr>
<td>stxvrhx</td>
<td>Store VSX Vector Rightmost Halfword Indexed</td>
<td>652</td>
</tr>
</tbody>
</table>

Table 15. VSX Vector Store Rightmost Element Instructions
### Table 15. VSX Vector Store Rightmost Element Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stxvrwx</td>
<td>Store VSX Vector Rightmost Word Indexed</td>
<td>652</td>
</tr>
</tbody>
</table>

### Table 16. VSX Vector Store with Length Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stxvl</td>
<td>Store VSX Vector with Length</td>
<td>648</td>
</tr>
<tr>
<td>stxvll</td>
<td>Store VSX Vector with Length Left-justified</td>
<td>650</td>
</tr>
</tbody>
</table>

### 7.6.1.1.3 VSX Vector Paired Storage Access Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxvp</td>
<td>Load VSX Vector Paired</td>
<td>625</td>
</tr>
<tr>
<td>lxvpx</td>
<td>Load VSX Vector Paired Indexed</td>
<td>626</td>
</tr>
<tr>
<td>plxvp</td>
<td>Prefixed Load VSX Vector Paired</td>
<td>625</td>
</tr>
</tbody>
</table>

### Table 17. VSX Vector Paired Load Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stxvp</td>
<td>Store VSX Vector Paired</td>
<td>654</td>
</tr>
<tr>
<td>stxvpx</td>
<td>Store VSX Vector Paired Indexed</td>
<td>655</td>
</tr>
<tr>
<td>pstxvp</td>
<td>Prefixed Store VSX Vector Paired</td>
<td>654</td>
</tr>
</tbody>
</table>

### Table 18. VSX Vector Paired Store Instructions
7.6.1.2 VSX Binary Floating-Point Sign Manipulation Instructions

7.6.1.2.1 VSX Scalar Binary Floating-Point Sign Manipulation Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsabsp</td>
<td>VSX Scalar Absolute Double-Precision</td>
<td>658</td>
</tr>
<tr>
<td>xsabssp</td>
<td>VSX Scalar Absolute Quad-Precision</td>
<td>658</td>
</tr>
<tr>
<td>xscpsgndp</td>
<td>VSX Scalar Copy Sign Double-Precision</td>
<td>682</td>
</tr>
<tr>
<td>xscpsgnqp</td>
<td>VSX Scalar Copy Sign Quad-Precision</td>
<td>682</td>
</tr>
<tr>
<td>xsnabsdp</td>
<td>VSX Scalar Negative Absolute Double-Precision</td>
<td>763</td>
</tr>
<tr>
<td>xsnabsqp</td>
<td>VSX Scalar Negative Absolute Quad-Precision</td>
<td>763</td>
</tr>
<tr>
<td>xsnegdp</td>
<td>VSX Scalar Negate Double-Precision</td>
<td>764</td>
</tr>
<tr>
<td>xsnegqp</td>
<td>VSX Scalar Negate Quad-Precision</td>
<td>764</td>
</tr>
</tbody>
</table>

Table 19. VSX Scalar BFP Sign Manipulation Instructions

7.6.1.2.2 VSX Vector Binary Floating-Point Sign Manipulation Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvabsp</td>
<td>VSX Vector Absolute Double-Precision</td>
<td>820</td>
</tr>
<tr>
<td>xvabssp</td>
<td>VSX Vector Absolute Single-Precision</td>
<td>820</td>
</tr>
<tr>
<td>xvcpsgndp</td>
<td>VSX Vector Copy Sign Double-Precision</td>
<td>838</td>
</tr>
<tr>
<td>xvcpsgnsp</td>
<td>VSX Vector Copy Sign Single-Precision</td>
<td>838</td>
</tr>
<tr>
<td>xvnabsdp</td>
<td>VSX Vector Negative Absolute Double-Precision</td>
<td>921</td>
</tr>
<tr>
<td>xvnabssp</td>
<td>VSX Vector Negative Absolute Single-Precision</td>
<td>921</td>
</tr>
<tr>
<td>xvnegdp</td>
<td>VSX Vector Negate Double-Precision</td>
<td>922</td>
</tr>
<tr>
<td>xvnegsp</td>
<td>VSX Vector Negate Single-Precision</td>
<td>922</td>
</tr>
</tbody>
</table>

Table 20. VSX Vector BFP Sign Manipulation Instructions

7.6.1.3 VSX Binary Floating-Point Arithmetic Instructions

7.6.1.3.1 VSX Scalar Binary Floating-Point Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsadddp</td>
<td>VSX Scalar Add Double-Precision</td>
<td>659</td>
</tr>
<tr>
<td>xsadddp[o]</td>
<td>VSX Scalar Add Quad-Precision [using round to Odd]</td>
<td>666</td>
</tr>
<tr>
<td>xsddspsp</td>
<td>VSX Scalar Add Single-Precision</td>
<td>664</td>
</tr>
<tr>
<td>xsdivdp</td>
<td>VSX Scalar Divide Double-Precision</td>
<td>717</td>
</tr>
<tr>
<td>xsdivqpsp</td>
<td>VSX Scalar Divide Quad-Precision [using round to Odd]</td>
<td>719</td>
</tr>
<tr>
<td>xsdivsp</td>
<td>VSX Scalar Divide Single-Precision</td>
<td>721</td>
</tr>
<tr>
<td>xsmulp</td>
<td>VSX Scalar Multiply Double-Precision</td>
<td>757</td>
</tr>
<tr>
<td>xsmulqpsp</td>
<td>VSX Scalar Multiply Quad-Precision [using round to Odd]</td>
<td>759</td>
</tr>
<tr>
<td>xsmulsp</td>
<td>VSX Scalar Multiply Single-Precision</td>
<td>761</td>
</tr>
<tr>
<td>xssqrtdp</td>
<td>VSX Scalar Square Root Double-Precision</td>
<td>801</td>
</tr>
<tr>
<td>xssqrtqpsp</td>
<td>VSX Scalar Square Root Quad-Precision [using round to Odd]</td>
<td>803</td>
</tr>
<tr>
<td>xssqrtsp</td>
<td>VSX Scalar Square Root Single-Precision</td>
<td>805</td>
</tr>
<tr>
<td>xsubdp</td>
<td>VSX Scalar Subtract Double-Precision</td>
<td>807</td>
</tr>
<tr>
<td>xssubqpsp</td>
<td>VSX Scalar Subtract Quad-Precision [using round to Odd]</td>
<td>809</td>
</tr>
<tr>
<td>xssubsp</td>
<td>VSX Scalar Subtract Single-Precision</td>
<td>811</td>
</tr>
</tbody>
</table>

Table 21. VSX Scalar BFP Elementary Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsmaddadp</td>
<td>VSX Scalar Multiply-Add Type-A Double-Precision</td>
<td>725</td>
</tr>
<tr>
<td>xsmaddasp</td>
<td>VSX Scalar Multiply-Add Type-A Single-Precision</td>
<td>728</td>
</tr>
</tbody>
</table>

Table 22. VSX Scalar BFP Multiply-Add-class Instructions
### Table 22. VSX Scalar BFP Multiply-Add-class Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsmsaddp</td>
<td>VSX Scalar Multiply-Add Type-M Double-Precision</td>
<td>725</td>
</tr>
<tr>
<td>xsmsadmp</td>
<td>VSX Scalar Multiply-Add Type-M Single-Precision</td>
<td>728</td>
</tr>
<tr>
<td>xsmsaddqo</td>
<td>VSX Scalar Multiply-Add Quad-Precision [using round to Odd]</td>
<td>731</td>
</tr>
<tr>
<td>xsmsubdp</td>
<td>VSX Scalar Multiply-Subtract Type-A Double-Precision</td>
<td>748</td>
</tr>
<tr>
<td>xsmsubasp</td>
<td>VSX Scalar Multiply-Subtract Type-A Single-Precision</td>
<td>751</td>
</tr>
<tr>
<td>xsmsubmip</td>
<td>VSX Scalar Multiply-Subtract Type-M Double-Precision</td>
<td>748</td>
</tr>
<tr>
<td>xsmsubmisp</td>
<td>VSX Scalar Multiply-Subtract Type-M Single-Precision</td>
<td>751</td>
</tr>
<tr>
<td>xsmsubcqo</td>
<td>VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd]</td>
<td>754</td>
</tr>
<tr>
<td>xsnmsaddp</td>
<td>VSX Scalar Negative Multiply-Add Type-A Double-Precision</td>
<td>765</td>
</tr>
<tr>
<td>xsnmsadsp</td>
<td>VSX Scalar Negative Multiply-Add Type-A Single-Precision</td>
<td>770</td>
</tr>
<tr>
<td>xsnmsmdbp</td>
<td>VSX Scalar Negative Multiply-Add Type-M Double-Precision</td>
<td>765</td>
</tr>
<tr>
<td>xsnmsmdbsp</td>
<td>VSX Scalar Negative Multiply-Add Type-M Single-Precision</td>
<td>770</td>
</tr>
<tr>
<td>xsnmsaddqo</td>
<td>VSX Scalar Negative Multiply-Add Quad-Precision [using round to Odd]</td>
<td>773</td>
</tr>
<tr>
<td>xsnmsubadp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Double-Precision</td>
<td>776</td>
</tr>
<tr>
<td>xsnmsubasp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Single-Precision</td>
<td>779</td>
</tr>
<tr>
<td>xsnmsusbmp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Double-Precision</td>
<td>776</td>
</tr>
<tr>
<td>xsnmsusbmsp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Single-Precision</td>
<td>779</td>
</tr>
<tr>
<td>xsnmsubcqo</td>
<td>VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd]</td>
<td>782</td>
</tr>
</tbody>
</table>

### Table 23. VSX Scalar Software BFP Divide/Square Root Instructions

#### 7.6.1.3.2 VSX Vector BFP Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvadddp</td>
<td>VSX Vector Add Double-Precision</td>
<td>821</td>
</tr>
<tr>
<td>xvadsp</td>
<td>VSX Vector Add Single-Precision</td>
<td>825</td>
</tr>
<tr>
<td>xvdvdp</td>
<td>VSX Vector Divide Double-Precision</td>
<td>867</td>
</tr>
<tr>
<td>xvdvsp</td>
<td>VSX Vector Divide Single-Precision</td>
<td>869</td>
</tr>
<tr>
<td>xvmulp</td>
<td>VSX Vector Multiply Double-Precision</td>
<td>917</td>
</tr>
<tr>
<td>xvmulp</td>
<td>VSX Vector Multiply Single-Precision</td>
<td>919</td>
</tr>
<tr>
<td>xvsqrtddp</td>
<td>VSX Vector Square Root Double-Precision</td>
<td>948</td>
</tr>
<tr>
<td>xvsqrtsp</td>
<td>VSX Vector Square Root Single-Precision</td>
<td>949</td>
</tr>
<tr>
<td>xvsusbdp</td>
<td>VSX Vector Subtract Double-Precision</td>
<td>950</td>
</tr>
<tr>
<td>xvsusbpsp</td>
<td>VSX Vector Subtract Single-Precision</td>
<td>952</td>
</tr>
</tbody>
</table>

### Table 24. VSX Vector BFP Elementary Arithmetic Instructions
### VSX Vector BFP Multiply-Add-class Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvredp</td>
<td>VSX Vector Reciprocal Estimate Double-Precision</td>
<td>940</td>
</tr>
<tr>
<td>xresp</td>
<td>VSX Vector Reciprocal Estimate Single-Precision</td>
<td>941</td>
</tr>
<tr>
<td>xrvsqrtedp</td>
<td>VSX Vector Reciprocal Square Root Estimate Double-Precision</td>
<td>946</td>
</tr>
<tr>
<td>xrvsqrtesp</td>
<td>VSX Vector Reciprocal Square Root Estimate Single-Precision</td>
<td>947</td>
</tr>
<tr>
<td>xtdivdp</td>
<td>VSX Vector Test for software Divide Double-Precision</td>
<td>954</td>
</tr>
<tr>
<td>xtdivsp</td>
<td>VSX Vector Test for software Divide Single-Precision</td>
<td>955</td>
</tr>
<tr>
<td>xtsqrtdp</td>
<td>VSX Vector Test for software Square Root Double-Precision</td>
<td>956</td>
</tr>
<tr>
<td>xtsqrtsp</td>
<td>VSX Vector Test for software Square Root Single-Precision</td>
<td>956</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvmaddadp</td>
<td>VSX Vector Multiply-Add Type-A Double-Precision</td>
<td>897</td>
</tr>
<tr>
<td>xvmaddasp</td>
<td>VSX Vector Multiply-Add Type-A Single-Precision</td>
<td>900</td>
</tr>
<tr>
<td>xvmaddmdp</td>
<td>VSX Vector Multiply-Add Type-M Double-Precision</td>
<td>897</td>
</tr>
<tr>
<td>xvmaddmsp</td>
<td>VSX Vector Multiply-Add Type-M Single-Precision</td>
<td>900</td>
</tr>
<tr>
<td>xvmsubadp</td>
<td>VSX Vector Multiply-Subtract Type-A Double-Precision</td>
<td>911</td>
</tr>
<tr>
<td>xvmsubasp</td>
<td>VSX Vector Multiply-Subtract Type-A Single-Precision</td>
<td>914</td>
</tr>
<tr>
<td>xvmsubmdp</td>
<td>VSX Vector Multiply-Subtract Type-M Double-Precision</td>
<td>911</td>
</tr>
<tr>
<td>xvmsubmsp</td>
<td>VSX Vector Multiply-Subtract Type-M Single-Precision</td>
<td>914</td>
</tr>
<tr>
<td>xvnmsubadp</td>
<td>VSX Vector Negative Multiply-Add Type-A Double-Precision</td>
<td>923</td>
</tr>
<tr>
<td>xvnmsaddsp</td>
<td>VSX Vector Negative Multiply-Add Type-A Single-Precision</td>
<td>927</td>
</tr>
<tr>
<td>xvnmsaddmdp</td>
<td>VSX Vector Negative Multiply-Add Type-M Double-Precision</td>
<td>923</td>
</tr>
<tr>
<td>xvnmsaddmsp</td>
<td>VSX Vector Negative Multiply-Add Type-M Single-Precision</td>
<td>927</td>
</tr>
<tr>
<td>xvnmsubadp</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Double-Precision</td>
<td>930</td>
</tr>
<tr>
<td>xvnmsubasp</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Single-Precision</td>
<td>933</td>
</tr>
<tr>
<td>xvnmsubmdp</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Double-Precision</td>
<td>930</td>
</tr>
<tr>
<td>xvnmsubmsp</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Single-Precision</td>
<td>933</td>
</tr>
</tbody>
</table>

### VSX Vector BFP Software Divide/Square Root Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvredp</td>
<td>VSX Vector Reciprocal Estimate Double-Precision</td>
<td>940</td>
</tr>
<tr>
<td>xresp</td>
<td>VSX Vector Reciprocal Estimate Single-Precision</td>
<td>941</td>
</tr>
<tr>
<td>xrvsqrtedp</td>
<td>VSX Vector Reciprocal Square Root Estimate Double-Precision</td>
<td>946</td>
</tr>
<tr>
<td>xrvsqrtesp</td>
<td>VSX Vector Reciprocal Square Root Estimate Single-Precision</td>
<td>947</td>
</tr>
<tr>
<td>xtdivdp</td>
<td>VSX Vector Test for software Divide Double-Precision</td>
<td>954</td>
</tr>
<tr>
<td>xtdivsp</td>
<td>VSX Vector Test for software Divide Single-Precision</td>
<td>955</td>
</tr>
<tr>
<td>xtsqrtdp</td>
<td>VSX Vector Test for software Square Root Double-Precision</td>
<td>956</td>
</tr>
<tr>
<td>xtsqrtsp</td>
<td>VSX Vector Test for software Square Root Single-Precision</td>
<td>956</td>
</tr>
</tbody>
</table>
7.6.1.4 VSX Binary Floating-Point Compare Instructions

7.6.1.4.1 VSX Scalar BFP Compare Instructions

Table 27. VSX Scalar BFP Compare Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscmpodp</td>
<td>VSX Scalar Compare Ordered Double-Precision</td>
<td>676</td>
</tr>
<tr>
<td>xscmpoqp</td>
<td>VSX Scalar Compare Ordered Quad-Precision</td>
<td>678</td>
</tr>
<tr>
<td>xscmpudp</td>
<td>VSX Scalar Compare Unordered Double-Precision</td>
<td>679</td>
</tr>
<tr>
<td>xscmpuqp</td>
<td>VSX Scalar Compare Unordered Quad-Precision</td>
<td>681</td>
</tr>
</tbody>
</table>

Table 28. VSX Scalar BFP Predicate Compare Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscmpeqdp</td>
<td>VSX Scalar Compare Equal Double-Precision</td>
<td>670</td>
</tr>
<tr>
<td>xscmpeqqp</td>
<td>VSX Scalar Compare Equal Quad-Precision</td>
<td>671</td>
</tr>
<tr>
<td>xscmpgedp</td>
<td>VSX Scalar Compare Greater Than or Equal Double-Precision</td>
<td>672</td>
</tr>
<tr>
<td>xscmpgeqp</td>
<td>VSX Scalar Compare Greater Than or Equal Quad-Precision</td>
<td>673</td>
</tr>
<tr>
<td>xscmpgtdp</td>
<td>VSX Scalar Compare Greater Than Double-Precision</td>
<td>674</td>
</tr>
<tr>
<td>xscmpgtqp</td>
<td>VSX Scalar Compare Greater Than Quad-Precision</td>
<td>675</td>
</tr>
</tbody>
</table>

Table 29. VSX Scalar BFP Maximum/Minimum Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsmaxcdp</td>
<td>VSX Scalar Maximum Type-C Double-Precision</td>
<td>736</td>
</tr>
<tr>
<td>xsmaxcq</td>
<td>VSX Scalar Maximum Type-C Quad-Precision</td>
<td>738</td>
</tr>
<tr>
<td>xsmaxd</td>
<td>VSX Scalar Maximum Double-Precision</td>
<td>734</td>
</tr>
<tr>
<td>xsmaxjd</td>
<td>VSX Scalar Maximum Type-J Double-Precision</td>
<td>739</td>
</tr>
<tr>
<td>xsmincdp</td>
<td>VSX Scalar Minimum Type-C Double-Precision</td>
<td>743</td>
</tr>
<tr>
<td>xsmincq</td>
<td>VSX Scalar Minimum Type-C Quad-Precision</td>
<td>745</td>
</tr>
<tr>
<td>xsmindp</td>
<td>VSX Scalar Minimum Double-Precision</td>
<td>741</td>
</tr>
<tr>
<td>xsminjd</td>
<td>VSX Scalar Minimum Type-J Double-Precision</td>
<td>746</td>
</tr>
</tbody>
</table>

7.6.1.4.2 VSX Vector BFP Compare Instructions

Table 30. VSX Vector BFP Predicate Compare Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvcmpeqdp</td>
<td>VSX Vector Compare Equal To Double-Precision</td>
<td>832</td>
</tr>
<tr>
<td>xvcmpeqsp</td>
<td>VSX Vector Compare Equal To Single-Precision</td>
<td>833</td>
</tr>
<tr>
<td>xvcmpgedp</td>
<td>VSX Vector Compare Greater Than or Equal To Double-Precision</td>
<td>834</td>
</tr>
<tr>
<td>xvcmpgesp</td>
<td>VSX Vector Compare Greater Than or Equal To Single-Precision</td>
<td>835</td>
</tr>
<tr>
<td>xvcmpgtdp</td>
<td>VSX Vector Compare Greater Than Double-Precision</td>
<td>836</td>
</tr>
<tr>
<td>xvcmpgtsp</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td>837</td>
</tr>
</tbody>
</table>

Table 31. VSX Vector BFP Maximum/Minimum Instructions
7.6.1.5 VSX Binary Floating-Point Round to Shorter Precision Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsrqpxp</td>
<td>VSX Scalar Round Quad-Precision to Double-Extended-Precision</td>
<td>795</td>
</tr>
<tr>
<td>xsrp</td>
<td>VSX Scalar Round to Single-Precision</td>
<td>797</td>
</tr>
</tbody>
</table>

Table 32.VSX Scalar BFP Round to Shorter Precision Instructions

7.6.1.6 VSX Binary Floating-Point Convert to Shorter Precision Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscvdphp</td>
<td>VSX Scalar Convert with round Double-Precision to Half-Precision format</td>
<td>683</td>
</tr>
<tr>
<td>xscvdpsp</td>
<td>VSX Scalar Convert with round Double-Precision to Single-Precision format</td>
<td>685</td>
</tr>
<tr>
<td>xscvdpspn</td>
<td>VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling</td>
<td>686</td>
</tr>
<tr>
<td>xcvqdpdp[o]</td>
<td>VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd]</td>
<td>797</td>
</tr>
</tbody>
</table>

Table 33.VSX Scalar BFP Convert to Shorter Precision Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcvbf16sp</td>
<td>VSX Vector Convert bfloat16 to Single-Precision format</td>
<td>839</td>
</tr>
<tr>
<td>xcvdpssp</td>
<td>VSX Vector Convert with round Double-Precision to Single-Precision format</td>
<td>840</td>
</tr>
<tr>
<td>xcvspshp</td>
<td>VSX Vector Convert with round Single-Precision to Half-Precision format</td>
<td>852</td>
</tr>
</tbody>
</table>

Table 34.VSX Vector BFP Convert to Shorter Precision Instructions

7.6.1.7 VSX Binary Floating-Point Convert to Longer Precision Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscvdqpp</td>
<td>VSX Scalar Convert Double-Precision to Quad-Precision format</td>
<td>684</td>
</tr>
<tr>
<td>xscvhpdp</td>
<td>VSX Scalar Convert Half-Precision to Double-Precision format</td>
<td>695</td>
</tr>
<tr>
<td>xscvsdpdp</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format</td>
<td>709</td>
</tr>
<tr>
<td>xscvsdpdn</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling</td>
<td>710</td>
</tr>
</tbody>
</table>

Table 35.VSX Scalar BFP Convert to Longer Precision Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcvhpssp</td>
<td>VSX Vector Convert Half-Precision to Single-Precision format</td>
<td>849</td>
</tr>
<tr>
<td>xcvspbf16</td>
<td>VSX Vector Convert with round Single-Precision to bfloat16 format</td>
<td>850</td>
</tr>
<tr>
<td>xcvspdp</td>
<td>VSX Vector Convert Single-Precision to Double-Precision format</td>
<td>851</td>
</tr>
</tbody>
</table>

Table 36.VSX Vector BFP Convert to Longer Precision Instructions
### 7.6.1.8 VSX Binary Floating-Point Round to Integral Instructions

#### 7.6.1.8.1 VSX Scalar BFP Round to Integral Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsrdpi</td>
<td>VSX Scalar Round to Double-Precision Integer using round to Nearest Away</td>
<td>785</td>
</tr>
<tr>
<td>xsrdpic</td>
<td>VSX Scalar Round to Double-Precision Integer exact using Current rounding mode</td>
<td>786</td>
</tr>
<tr>
<td>xsrdpim</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward -Infinity</td>
<td>787</td>
</tr>
<tr>
<td>xsrdpip</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward +Infinity</td>
<td>788</td>
</tr>
<tr>
<td>xsrdpiz</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward Zero</td>
<td>789</td>
</tr>
<tr>
<td>xsrqpi</td>
<td>VSX Scalar Round to Quad-Precision Integer</td>
<td>793</td>
</tr>
<tr>
<td>xsrqpix</td>
<td>VSX Scalar Round to Quad-Precision Integer with Inexact</td>
<td>793</td>
</tr>
<tr>
<td>xvrdpi</td>
<td>VSX Vector Round to Double-Precision Integer using round to Nearest Away</td>
<td>936</td>
</tr>
<tr>
<td>xvrdpic</td>
<td>VSX Vector Round to Double-Precision Integer Exact using Current rounding mode</td>
<td>937</td>
</tr>
<tr>
<td>xvrdpim</td>
<td>VSX Vector Round to Double-Precision Integer using round toward -Infinity</td>
<td>938</td>
</tr>
<tr>
<td>xvrdpip</td>
<td>VSX Vector Round to Double-Precision Integer using round toward +Infinity</td>
<td>939</td>
</tr>
<tr>
<td>xvrdpiz</td>
<td>VSX Vector Round to Double-Precision Integer using round toward Zero</td>
<td>939</td>
</tr>
</tbody>
</table>

Table 37. VSX Scalar BFP Round to Integral Instructions

#### 7.6.1.8.2 VSX Vector BFP Round to Integral Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvrspi</td>
<td>VSX Vector Round to Single-Precision Integer using round to Nearest Away</td>
<td>942</td>
</tr>
<tr>
<td>xvrsptic</td>
<td>VSX Vector Round to Single-Precision Integer Exact using Current rounding mode</td>
<td>943</td>
</tr>
<tr>
<td>xvrsptic</td>
<td>VSX Vector Round to Single-Precision Integer using round toward -Infinity</td>
<td>944</td>
</tr>
<tr>
<td>xvrspip</td>
<td>VSX Vector Round to Single-Precision Integer using round toward +Infinity</td>
<td>945</td>
</tr>
<tr>
<td>xvrspiz</td>
<td>VSX Vector Round to Single-Precision Integer using round toward Zero</td>
<td>945</td>
</tr>
</tbody>
</table>

Table 38. VSX Vector BFP Round to Integral Instructions

### 7.6.1.9 VSX Binary Floating-Point Convert To Integer Instructions

#### 7.6.1.9.1 VSX Scalar BFP Convert To Integer Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscvdpsxds</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format</td>
<td>687</td>
</tr>
<tr>
<td>xscvdpxwes</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Word format</td>
<td>689</td>
</tr>
<tr>
<td>xscvdpxwes</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format</td>
<td>691</td>
</tr>
<tr>
<td>xscvdpxwse</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format</td>
<td>693</td>
</tr>
<tr>
<td>xscvdpsdz</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format</td>
<td>697</td>
</tr>
<tr>
<td>xscvdpsqz</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Quadword</td>
<td>699</td>
</tr>
<tr>
<td>xscvdpswz</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format</td>
<td>701</td>
</tr>
<tr>
<td>xscvdpdz</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format</td>
<td>703</td>
</tr>
<tr>
<td>xscvdpqz</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Unsigned Quadword</td>
<td>705</td>
</tr>
<tr>
<td>xscvdpwz</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format</td>
<td>707</td>
</tr>
</tbody>
</table>

Table 39. VSX Scalar BFP Convert to Integer Instructions
7.6.1.9.2 VSX Vector BFP Convert To Integer Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvcvdpxsxsds</td>
<td>VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format</td>
<td>841</td>
</tr>
<tr>
<td>xvcvdpxsxs</td>
<td>VSX Vector Convert with round to zero Double-Precision to Signed Word format</td>
<td>843</td>
</tr>
<tr>
<td>xvcvdpxudxs</td>
<td>VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format</td>
<td>845</td>
</tr>
<tr>
<td>xvcvdpuxws</td>
<td>VSX Vector Convert with round to zero Double-Precision to Unsigned Word format</td>
<td>847</td>
</tr>
<tr>
<td>xvcvpspxxsd</td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format</td>
<td>853</td>
</tr>
<tr>
<td>xvcvpspxss</td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Word format</td>
<td>855</td>
</tr>
<tr>
<td>xvcvpspxudxs</td>
<td>VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format</td>
<td>857</td>
</tr>
<tr>
<td>xvcvpspxws</td>
<td>VSX Vector Convert with round to zero Single-Precision to Unsigned Word format</td>
<td>859</td>
</tr>
</tbody>
</table>

Table 40. VSX Vector BFP Convert To Integer Instructions

7.6.1.10 VSX Binary Floating-Point Convert From Integer Instructions

7.6.1.10.1 VSX Scalar BFP Convert From Integer Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscvsvdqdp</td>
<td>VSX Scalar Convert Signed Doubleword to Quad-Precision format</td>
<td>714</td>
</tr>
<tr>
<td>xscvsvsqqp</td>
<td>VSX Scalar Convert with round Signed Quadword to Quad-Precision</td>
<td>711</td>
</tr>
<tr>
<td>xscvsvxdqdp</td>
<td>VSX Scalar Convert with round Signed Doubleword to Double-Precision format</td>
<td>712</td>
</tr>
<tr>
<td>xscvsvxsdp</td>
<td>VSX Scalar Convert with round Signed Doubleword to Single-Precision format</td>
<td>713</td>
</tr>
<tr>
<td>xscvsvsdqp</td>
<td>VSX Scalar Convert Signed Doubleword to Quad-Precision format</td>
<td>714</td>
</tr>
<tr>
<td>xscvsvsqqp</td>
<td>VSX Scalar Convert with round Signed Quadword to Quad-Precision</td>
<td>715</td>
</tr>
<tr>
<td>xscvsvxdqdp</td>
<td>VSX Scalar Convert with round Signed Doubleword to Double-Precision format</td>
<td>715</td>
</tr>
<tr>
<td>xscvsvxsdp</td>
<td>VSX Scalar Convert with round Signed Doubleword to Single-Precision format</td>
<td>716</td>
</tr>
</tbody>
</table>

Table 41. VSX Scalar BFP Convert from Integer Instructions

7.6.1.10.2 VSX Vector BFP Convert From Integer Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvcvsvxdqdp</td>
<td>VSX Vector Convert with round Signed Doubleword to Double-Precision format</td>
<td>861</td>
</tr>
<tr>
<td>xvcvsvxdqds</td>
<td>VSX Vector Convert with round Signed Doubleword to Double-Precision format</td>
<td>863</td>
</tr>
<tr>
<td>xvcvsvxdqdp</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Double-Precision format</td>
<td>864</td>
</tr>
<tr>
<td>xvcvsvxdqds</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Double-Precision format</td>
<td>866</td>
</tr>
<tr>
<td>xvcvsvxdqdp</td>
<td>VSX Vector Convert with round Signed Doubleword to Single-Precision format</td>
<td>862</td>
</tr>
<tr>
<td>xvcvsvxdqds</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Single-Precision format</td>
<td>865</td>
</tr>
<tr>
<td>xvcvsvxdqds</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Single-Precision format</td>
<td>866</td>
</tr>
</tbody>
</table>

Table 42. VSX Vector BFP Convert From Integer Instructions

7.6.1.11 VSX Binary Floating-Point Math Support Instructions

7.6.1.11.1 VSX Scalar BFP Math Support Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscmpexdpdp</td>
<td>VSX Scalar Compare Exponents Double-Precision</td>
<td>668</td>
</tr>
<tr>
<td>xscmpexpqdp</td>
<td>VSX Scalar Compare Exponents Quad-Precision</td>
<td>669</td>
</tr>
<tr>
<td>xsiexdpdp</td>
<td>VSX Scalar Insert Exponent Double-Precision</td>
<td>723</td>
</tr>
<tr>
<td>xsiexpqdp</td>
<td>VSX Scalar Insert Exponent Quad-Precision</td>
<td>724</td>
</tr>
<tr>
<td>xststdcdp</td>
<td>VSX Scalar Test Data Class Double-Precision</td>
<td>815</td>
</tr>
<tr>
<td>xststdcpp</td>
<td>VSX Scalar Test Data Class Quad-Precision</td>
<td>816</td>
</tr>
<tr>
<td>xststdcsp</td>
<td>VSX Scalar Test Data Class Single-Precision</td>
<td>817</td>
</tr>
</tbody>
</table>

Table 43. VSX Scalar BFP Math Support Instructions
7.6.11.2 VSX Vector BFP Math Support Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsvexpdp</td>
<td>VSX Scalar Extract Exponent Double-Precision</td>
<td>818</td>
</tr>
<tr>
<td>xsvexpqp</td>
<td>VSX Scalar Extract Exponent Quad-Precision</td>
<td>818</td>
</tr>
<tr>
<td>xsvsigidp</td>
<td>VSX Scalar Extract Significand Double-Precision</td>
<td>819</td>
</tr>
<tr>
<td>xsvsigidqp</td>
<td>VSX Scalar Extract Significand Quad-Precision</td>
<td>819</td>
</tr>
</tbody>
</table>

Table 43. VSX Scalar BFP Math Support Instructions

7.6.1.12 VSX Matrix-Multiply Assist (MMA) Instructions

The MMA facility is optional. Software that uses this facility should test for its availability and provide an alternate execution path.

7.6.1.12.1 VSX Accumulator Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsvxger2</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update)</td>
<td>891</td>
</tr>
<tr>
<td>xsvxger2pp</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate</td>
<td>891</td>
</tr>
<tr>
<td>xsvxger2s</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation</td>
<td>891</td>
</tr>
<tr>
<td>xsvxger2pp</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate</td>
<td>891</td>
</tr>
<tr>
<td>xsvxger8</td>
<td>VSX Vector 4-bit Signed Integer GER (rank-8 update)</td>
<td>883</td>
</tr>
<tr>
<td>xsvxger8pp</td>
<td>VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate</td>
<td>883</td>
</tr>
<tr>
<td>xsvxger4</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update)</td>
<td>886</td>
</tr>
<tr>
<td>xsvxger4pp</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate</td>
<td>886</td>
</tr>
</tbody>
</table>

Table 45. VSX Accumulator Move Instructions

7.6.1.12.2 VSX Binary Integer Outer-Product Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pmxv16ger2</td>
<td>Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update)</td>
<td>893</td>
</tr>
<tr>
<td>pmxv16ger2pp</td>
<td>Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate</td>
<td>893</td>
</tr>
<tr>
<td>pmxv16ger2s</td>
<td>Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation</td>
<td>893</td>
</tr>
<tr>
<td>pmxv16ger2pp</td>
<td>Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate</td>
<td>893</td>
</tr>
<tr>
<td>pmxv16ger8</td>
<td>Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update)</td>
<td>883</td>
</tr>
<tr>
<td>pmxv16ger8pp</td>
<td>Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate</td>
<td>883</td>
</tr>
<tr>
<td>pmxv16ger4</td>
<td>Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update)</td>
<td>886</td>
</tr>
<tr>
<td>pmxv16ger4pp</td>
<td>Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate</td>
<td>886</td>
</tr>
<tr>
<td>xvi16ger2</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update)</td>
<td>891</td>
</tr>
<tr>
<td>xvi16ger2pp</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate</td>
<td>891</td>
</tr>
<tr>
<td>xvi16ger2s</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation</td>
<td>893</td>
</tr>
</tbody>
</table>

Table 46. VSX Binary Integer Outer-Product Instructions
7.6.1.12.3 VSX Binary Floating-Point Outer-Product Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pmxvb16ger2</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (rank-2 update)</td>
<td>827</td>
</tr>
<tr>
<td>pmxvb16ger2nn</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate</td>
<td>827</td>
</tr>
<tr>
<td>pmxvb16ger2np</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate</td>
<td>827</td>
</tr>
<tr>
<td>pmxvb16ger2pn</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate</td>
<td>827</td>
</tr>
<tr>
<td>pmxvb16ger2pp</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate</td>
<td>827</td>
</tr>
<tr>
<td>pmxvf32ger</td>
<td>Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update)</td>
<td>875</td>
</tr>
<tr>
<td>pmxvf32gernn</td>
<td>Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate</td>
<td>875</td>
</tr>
<tr>
<td>pmxvf32gerpp</td>
<td>Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate</td>
<td>875</td>
</tr>
<tr>
<td>pmxvf64ger</td>
<td>Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update)</td>
<td>879</td>
</tr>
<tr>
<td>pmxvf64gernn</td>
<td>Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate</td>
<td>879</td>
</tr>
<tr>
<td>pmxvf64gerpp</td>
<td>Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate</td>
<td>879</td>
</tr>
</tbody>
</table>

Table 47. VSX Binary Floating-Point Outer-Product Instructions
### Table 47. VSX Binary Floating-Point Outer-Product Instructions (Continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvf16ger2</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update)</td>
<td>871</td>
</tr>
<tr>
<td>xvf16ger2nn</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate</td>
<td>871</td>
</tr>
<tr>
<td>xvf16ger2np</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate</td>
<td>871</td>
</tr>
<tr>
<td>xvf16ger2pn</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate</td>
<td>871</td>
</tr>
<tr>
<td>xvflger2pp</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate</td>
<td>871</td>
</tr>
<tr>
<td>xvf32ger</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update)</td>
<td>875</td>
</tr>
<tr>
<td>xvf32gernn</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate</td>
<td>875</td>
</tr>
<tr>
<td>xvf32gernp</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate</td>
<td>875</td>
</tr>
<tr>
<td>xvf32germp</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate</td>
<td>875</td>
</tr>
<tr>
<td>xvf32gerpn</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate</td>
<td>875</td>
</tr>
<tr>
<td>xvf64ger</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update)</td>
<td>879</td>
</tr>
<tr>
<td>xvf64gernn</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate</td>
<td>879</td>
</tr>
<tr>
<td>xvf64gerpp</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate</td>
<td>879</td>
</tr>
<tr>
<td>xvf64gerpn</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate</td>
<td>879</td>
</tr>
<tr>
<td>xvf64gerpp</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate</td>
<td>879</td>
</tr>
</tbody>
</table>

7.6.1.13 VSX Vector Logical Instructions

7.6.1.13.1 VSX Vector Logical Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxland</td>
<td>VSX Vector Logical AND</td>
<td>978</td>
</tr>
<tr>
<td>xxlandc</td>
<td>VSX Vector Logical AND with Complement</td>
<td>978</td>
</tr>
<tr>
<td>xxleqv</td>
<td>VSX Vector Logical Equivalence</td>
<td>979</td>
</tr>
<tr>
<td>xxlnand</td>
<td>VSX Vector Logical NAND</td>
<td>979</td>
</tr>
<tr>
<td>xxlnor</td>
<td>VSX Vector Logical NOR</td>
<td>980</td>
</tr>
<tr>
<td>xxlor</td>
<td>VSX Vector Logical OR</td>
<td>981</td>
</tr>
<tr>
<td>xxlorc</td>
<td>VSX Vector Logical OR with Complement</td>
<td>981</td>
</tr>
<tr>
<td>xxlxor</td>
<td>VSX Vector Logical XOR</td>
<td>981</td>
</tr>
</tbody>
</table>

Table 48. VSX Logical Instructions

7.6.1.13.2 VSX Vector Select Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxssel</td>
<td>VSX Vector Select</td>
<td>988</td>
</tr>
</tbody>
</table>

Table 49. VSX Vector Select Instruction

7.6.1.13.3 VSX Vector Evaluate Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxeval</td>
<td>VSX Vector Evaluate</td>
<td>967</td>
</tr>
</tbody>
</table>

Table 50. VSX Vector Select Instruction
7.6.1.13.4 VSX Vector Blend Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxbldvbd</td>
<td>VSX Vector Blend Variable Byte</td>
<td>962</td>
</tr>
<tr>
<td>xxbldvd</td>
<td>VSX Vector Blend Variable Doubleword</td>
<td>963</td>
</tr>
<tr>
<td>xxbldvh</td>
<td>VSX Vector Blend Variable Halfword</td>
<td>962</td>
</tr>
<tr>
<td>xxbldvw</td>
<td>VSX Vector Blend Variable Word</td>
<td>963</td>
</tr>
</tbody>
</table>

Table 51. VSX Vector Select Instruction

7.6.1.14 VSX Vector Permute-class Instructions

7.6.1.14.1 VSX Vector Byte-Reverse Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxbbrd</td>
<td>VSX Vector Byte-Reverse Doubleword</td>
<td>964</td>
</tr>
<tr>
<td>xxbbrh</td>
<td>VSX Vector Byte-Reverse Halfword</td>
<td>965</td>
</tr>
<tr>
<td>xxbbrq</td>
<td>VSX Vector Byte-Reverse Quadword</td>
<td>966</td>
</tr>
<tr>
<td>xxbbrw</td>
<td>VSX Vector Byte-Reverse Word</td>
<td>967</td>
</tr>
</tbody>
</table>

Table 52. VSX Vector Byte-Reverse Instructions

7.6.1.14.2 VSX Vector Insert/Extract Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxexctuw</td>
<td>VSX Vector Extract Unsigned Word</td>
<td>969</td>
</tr>
<tr>
<td>xxinsrtw</td>
<td>VSX Vector Insert Word</td>
<td>969</td>
</tr>
</tbody>
</table>

Table 53. VSX Vector Insert/Extract Instructions

7.6.1.14.3 VSX Vector Merge Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxmrghw</td>
<td>VSX Vector Merge High Word</td>
<td>982</td>
</tr>
<tr>
<td>xxmrglw</td>
<td>VSX Vector Merge Low Word</td>
<td>982</td>
</tr>
</tbody>
</table>

Table 54. VSX Vector Merge Instructions

7.6.1.14.4 VSX Vector Splat Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxsplit32dx</td>
<td>VSX Vector Splat Immediate32 Doubleword Indexed</td>
<td>992</td>
</tr>
<tr>
<td>xxsplitb</td>
<td>VSX Vector Splat Immediate Byte</td>
<td>991</td>
</tr>
<tr>
<td>xxsplitdp</td>
<td>VSX Vector Splat Immediate Double-Precision</td>
<td>991</td>
</tr>
<tr>
<td>xxsplitw</td>
<td>VSX Vector Splat Immediate Word</td>
<td>992</td>
</tr>
<tr>
<td>xxsplitw</td>
<td>VSX Vector Splat Word</td>
<td>993</td>
</tr>
</tbody>
</table>

Table 55. VSX Vector Splat Instructions

7.6.1.14.5 VSX Vector Permute Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxpermld</td>
<td>VSX Vector Permute Doubleword Immediate</td>
<td>986</td>
</tr>
<tr>
<td>xxperm</td>
<td>VSX Vector Permute</td>
<td>985</td>
</tr>
<tr>
<td>xxpermrd</td>
<td>VSX Vector Permute Right-indexed</td>
<td>985</td>
</tr>
<tr>
<td>xxpermx</td>
<td>VSX Vector Permute Extended</td>
<td>987</td>
</tr>
</tbody>
</table>

Table 56. VSX Vector Permute Instruction
7.6.1.14.6 VSX Vector Shift Left Double Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxslldwi</td>
<td>VSX Vector Shift Left Double by Word Immediate</td>
<td>990</td>
</tr>
</tbody>
</table>

Table 57. VSX Vector Shift Left Double Instruction

7.6.1.14.7 VSX Vector Generate Permute Control Vector Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxgenpcvbm</td>
<td>VSX Vector Generate PCV from Byte Mask</td>
<td>970</td>
</tr>
<tr>
<td>xxgenpcvdm</td>
<td>VSX Vector Generate PCV from Doubleword Mask</td>
<td>976</td>
</tr>
<tr>
<td>xxgenpcvhm</td>
<td>VSX Vector Generate PCV from Halfword Mask</td>
<td>972</td>
</tr>
<tr>
<td>xxgenpcvwm</td>
<td>VSX Vector Generate PCV from Word Mask</td>
<td>974</td>
</tr>
</tbody>
</table>

Table 58. VSX Vector Permute Control Vector Generate Instruction

7.6.1.15 VSX Vector Load Special Value Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxvkq</td>
<td>VSX Vector Load Special Value Quadword</td>
<td>620</td>
</tr>
</tbody>
</table>

Table 59. VSX Vector Load Special Value Instruction

7.6.1.16 VSX Vector Test Least-Significant Bit by Byte Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvtlsbb</td>
<td>VSX Vector Test Least-Significant Bit by Byte</td>
<td>959</td>
</tr>
</tbody>
</table>

Table 60. VSX Vector Load Special Value Instruction
7.6.2 VSX Instruction Description Conventions

7.6.2.1 VSX Instruction RTL Operators

\( x.\text{bit}[y] \)  
Return the contents of bit \( y \) of \( x \).

\( x.\text{bit}[y:z] \)  
Return the contents of bits \( y:z \) of \( x \).

\( x.\text{word}[y] \)  
Return the contents of word element \( y \) of \( x \).

\( x.\text{word}[y:z] \)  
Return the contents of word elements \( y:z \) of \( x \).

\( x.\text{dword}[y] \)  
Return the contents of doubleword element \( y \) of \( x \).

\( x.\text{dword}[y:z] \)  
Return the contents of doubleword elements \( y:z \) of \( x \).

\( x = y \)  
The value of \( y \) is placed into \( x \).

\( x \|\| y \)  
The value of \( y \) is ORed with the value \( x \) and placed into \( x \).

\( \neg x \)  
Return the one's complement of \( x \).

\( !x \)  
Return 1 if the contents of \( x \) are equal to 0, otherwise return 0.

\( x \|\| y \)  
Return the value of \( x \) concatenated with the value of \( y \). For example, \( 0b010 \|\| 0b111 \) is the same as \( 0b010111 \).

\( x ^ y \)  
Return the value of \( x \) exclusive ORed with the value of \( y \).

\( x ? y : z \)  
If the value of \( x \) is true, return the value of \( y \), otherwise return the value \( z \).

\( x+y \)  
\( x \) and \( y \) are integer values.

Return the sum of \( x \) and \( y \).
7.6.2.2 VSX Instruction RTL Function Calls

**bfloat16\_CONVERT\_FROM\_BFP(x)**

- x is a floating-point value represented in the working format.
  - If \(x\) is SNaN or QNaN, do the following.
    - Bit 0 of result is set to the value of \(x\) sign.
    - Bits 1:8 of result are set to the value 0b1111111.
    - Bits 9:15 of result are set to the value of bits 1:8 of \(x\) significand.
  - Otherwise, if \(x\) is an infinity, do the following.
    - Bit 0 of result is set to the value of \(x\) sign.
    - Bits 1:8 of result are set to the value 0b1111111.
    - Bits 9:15 of result are set to 0.
  - Otherwise, if \(x\) is a zero, do the following.
    - Bit 0 of result is set to the value of \(x\) sign.
    - Bits 9:15 of result are set to 0.
  - Otherwise, if \(x\) exponent is less than -126, do the following.
    - Bit 0 of result is set to the value of \(x\) sign.
    - \(sh\_cnt\) is set to the difference, -126 - \(x\) exponent.
    - Bits 1:8 of result are set to 0b00000000.
    - Bits 9:15 of result are set to bits 1:8 of \(x\) significand shifted right by \(sh\_cnt\) bits.
  - Otherwise, do the following.
    - Bit 0 of result is set to the value of \(x\) sign.
    - Bits 1:8 of result are set to the sum, \(x\) exponent + 127.
    - Bits 9:15 of result are set to bits 1:8 of \(x\) significand.

Return result (bfloat16 format).

**bfp\_ABSOLUTE(x)**

- x is a binary floating-point value represented in the binary floating-point working format.

Return x with sign set to 0.

**bfp\_ADD(x, y)**

- x is a binary floating-point value represented in the binary floating-point working format.
- y is a binary floating-point value represented in the binary floating-point working format.

- If \(x\) or \(y\) is an SNaN, \(vxsnan\_flag\) is set to 1.
- If \(x\) is an infinity and \(y\) is an infinity of the opposite sign, \(vxisi\_flag\) is set to 1.
- If \(x\) is a QNaN, return \(x\).
- Otherwise, if \(x\) is an SNaN, return \(x\) represented as a QNaN.
- Otherwise, if \(y\) is a QNaN, return \(y\).
- Otherwise, if \(y\) is an SNaN, return \(y\) represented as a QNaN.
- Otherwise, if \(x\) and \(y\) are infinities of opposite sign, return the standard QNaN.
- Otherwise, return the normalized sum of \(x\) and \(y\), having unbounded range and precision.

**bfp\_COMPARE\_EQ(x, y)**

- x is a binary floating-point value represented in the binary floating-point working format.
- y is a binary floating-point value represented in the binary floating-point working format.

- Return 0b00 if \(x\) is NaN or \(y\) is a NaN.
- Otherwise, return 0b1 if \(x\) is a Zero and \(y\) is a Zero.
- Otherwise, return 0b1 if \(x\) is equal to \(y\).
- Otherwise, return 0b0.
**bfp_COMPARE_GT(x, y)**

- x is a binary floating-point value represented in the binary floating-point working format.
- y is a binary floating-point value represented in the binary floating-point working format.

Return $0b0$ if x is NaN or y is a NaN.
Otherwise, return $0b0$ if x is a Zero and y is a Zero.
Otherwise, return $0b1$ if x is greater than y.
Otherwise, return $0b0$.

**bfp_COMPARE_LT(x, y)**

- x is a binary floating-point value represented in the binary floating-point working format.
- y is a binary floating-point value represented in the binary floating-point working format.

Return $0b0$ if x is NaN or y is a NaN.
Otherwise, return $0b0$ if x is a Zero and y is a Zero.
Otherwise, return $0b1$ if x is less than y.
Otherwise, return $0b0$.

**bfp_CONVERT_FROM_BFLOAT16(x)**

- x is a floating-point value represented in bfloat16 format.

Let sign be the contents of bit 0 of x.
Let exponent be the contents of bits 1:8 of x.
Let fraction be the contents of bits 9:15 of x.

Let result.sign be set to 0.
Let result.exponent be set to 0.
Let result.significand be set to 0.
Let result.class.SNaN be set to 0.
Let result.class.QNaN be set to 0.
Let result.class.Infinity be set to 0.
Let result.class.Zero be set to 0.
Let result.class.Denormal be set to 0.
Let result.class.Normal be set to 0.

If x is an SNaN, do the following.
- result.class.SNaN is set to 1.
- result.sign is set to the value of sign.
- The contents of result.significand are set to 0.
- The contents of bits 1:8 of result.significand are set to the value of fraction.

Otherwise, if x is a QNaN, do the following.
- result.class.QNaN is set to 1.
- result.sign is set to the value of sign.
- The contents of result.significand are set to 0.
- The contents of bits 1:8 of result.significand are set to the value of fraction.

Otherwise, if x is an Infinity value, do the following.
- result.class.Infinity is set to 1.
- result.sign is set to the value of sign.

Otherwise, if x is a Zero value, do the following.
- result.class.Zero is set to 1.
- result.sign is set to the value of sign.

Otherwise, if x is a Denormal value, do the following.
- result.class.Denormal is set to 1.
- result.sign is set to the value of sign.
result.exponent is set to the value -126.

The contents of bits 1:8 of result.significand are set to the value of fraction.

result.significand is shifted left until the contents bit 0 of result.significand are equal to 1.
result.exponent is decremented by the number of bits result.significand was shifted.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to the value of sign.
result.exponent is set to the value of exponent subtracted by 127.

The contents of bit 0 of result.significand are set to 1.
The contents of bits 1:8 of result.significand are set to the value of fraction.

Return result (binary floating-point working format).

bfp_CONVERT_FROM_BFP16(x)

x is a floating-point value represented in half-precision format.

Let exponent be the contents of bits 1:5 of x.
Let fraction be the contents of bits 6:15 of x.

Let result.sign be set to 0.
Let result.exponent be set to 0.
Let result.significand be set to 0.
Let result.class.SNaN be set to 0.
Let result.class.QNaN be set to 0.
Let result.class.Infinity be set to 0.
Let result.class.Zero be set to 0.
Let result.class.Denormal be set to 0.
Let result.class.Normal be set to 0.

If x is a SNaN, do the following.
result.class.SNaN is set to 1.
result.sign is set to the contents of bit 0 of x.

The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:10 of result.significand are set to the value of fraction.

Otherwise, if x is a QNaN, do the following.
result.class.QNaN is set to 1.
result.sign is set to the contents of bit 0 of x.

The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:10 of result.significand are set to the value of fraction.

Otherwise, if x is an Infinity value, do the following.
result.class.Infinity is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Zero value, do the following.
result.class.Zero is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Denormal value, do the following.
result.class.Denormal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exp is set to the value -14.
The contents of bit 0 of \texttt{result.significand} are set to 0.
The contents of bits 1:10 of \texttt{result.significand} are set to the value of \texttt{fraction}.
\texttt{result.significand} is shifted left until the contents bit 0 of \texttt{result.significand} are equal to 1.
\texttt{result.exponent} is decremented by the the number of bits \texttt{result.significand} was shifted.

Otherwise, do the following.
\texttt{result.class.Normal} is set to 1.
\texttt{result.sign} is set to the contents of bit 0 of \texttt{x}.
\texttt{result.exp} is set to the value of \texttt{exponent} subtracted by 15.

The contents of bit 0 of \texttt{result.significand} are set to 1.
The contents of bits 1:10 of \texttt{result.significand} are set to the value of \texttt{fraction}.

Return \texttt{result}.

\texttt{bfp\_CONVERT\_FROM\_BFP32(x)}
\texttt{x} is a floating-point value represented in single-precision format.

Let \texttt{exponent} be the contents of bits 1:8 of \texttt{x}.
Let \texttt{fraction} be the contents of bits 9:31 of \texttt{x}.

Let \texttt{result.sign} be initialized to 0.
Let \texttt{result.exponent} be initialized to 0.
Let \texttt{result.significand} be initialized to 0.
Let \texttt{result.class.SNaN} be initialized to 0.
Let \texttt{result.class.QNaN} be initialized to 0.
Let \texttt{result.class.Infinity} be initialized to 0.
Let \texttt{result.class.Zero} be initialized to 0.
Let \texttt{result.class.Denormal} be initialized to 0.
Let \texttt{result.class.Normal} be initialized to 0.

If \texttt{x} is a SNaN, do the following.
\texttt{result.class.SNaN} is set to 1.
\texttt{result.sign} is set to the contents of bit 0 of \texttt{x}.

The contents of bit 0 of \texttt{result.significand} are set to 0.
The contents of bits 1:23 of \texttt{result.significand} are set to the value of \texttt{fraction}.

Otherwise, if \texttt{x} is a QNaN, do the following.
\texttt{result.class.QNaN} is set to 1.
\texttt{result.sign} is set to the contents of bit 0 of \texttt{x}.

The contents of bit 0 of \texttt{result.significand} are set to 0.
The contents of bits 1:23 of \texttt{result.significand} are set to the value of \texttt{fraction}.

Otherwise, if \texttt{x} is an Infinity value, do the following.
\texttt{result.class.Infinity} is set to 1.
\texttt{result.sign} is set to the contents of bit 0 of \texttt{x}.

Otherwise, if \texttt{x} is a Zero value, do the following.
\texttt{result.class.Zero} is set to 1.
\texttt{result.sign} is set to the contents of bit 0 of \texttt{x}.

Otherwise, if \texttt{x} is a Denormal value, do the following.
\texttt{result.class.Denormal} is set to 1.
\texttt{result.sign} is set to the contents of bit 0 of \texttt{x}.
\texttt{result.exponent} is set to the value -126.

The contents of bit 0 of \texttt{result.significand} are set to 0.
The contents of bits 1:23 of result.significand are set to the value of fraction.

result.significand is shifted left until the contents bit 0 of result.significand are equal to 1. Result.exponent is decremented by the the number of bits result.significand was shifted.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exponent is set to the value of exponent subtracted by 127.

The contents of bit 0 of result.significand are set to 1.
The contents of bits 1:23 of result.significand are set to the value of fraction.

Return result.

bfp_CONVERT_FROM_BFP64(x)

x is a binary floating-point value represented in double-precision format.

Let exponent be the contents of bits 1:11 of x.
Let fraction be the contents of bits 12:63 of x.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is a SNaN, do the following.
result.class.SNaN is set to 1.
result.sign is set to the contents of bit 0 of x.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:52 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.

Otherwise, if x is a QNaN, do the following.
result.class.QNaN is set to 1.
result.sign is set to the contents of bit 0 of x.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:52 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.

Otherwise, if x is an Infinity, do the following.
result.class.Infinity is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Zero, do the following.
result.class.Zero is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Denormal, do the following.
result.class.Denormal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exp is set to the value -1022.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:52 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.
result.significand is shifted left until the contents bit 0 of result.significand are equal to 1.
result.exponent is decremented by the the number of bits result.significand was shifted.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exp is set to the value of exponent subtracted by 1023.
The contents of bit 0 of result.significand are set to 1.
The contents of bits 1:52 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.

Return result (i.e., the value x in the binary floating-point working format).

bfp_CONVERT_FROM_BFP128(x)
x is a binary floating-point value represented in quad-precision format.

Let exponent be the contents of bits 1:15 of x.
Let fraction be the contents of bits 16:127 of x.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is a SNaN, do the following.
result.class.SNaN is set to 1.
result.sign is set to the contents of bit 0 of x.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:112 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.

Otherwise, if x is a QNaN, do the following.
result.class.QNaN is set to 1.
result.sign is set to the contents of bit 0 of x.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:112 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.

Otherwise, if x is an Infinity, do the following.
result.class.Infinity is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Zero, do the following.
result.class.Zero is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Denormal, do the following.
result.class.Denormal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exp is set to the value -16382.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:112 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.
result.significand is shifted left until the contents bit 0 of result.significand are equal to 1.
result.exponent is decremented by the number of bits result.significand was shifted.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exp is set to the value of exponent subtracted by 16383.
The contents of bit 0 of result.significand are set to 1.
The contents of bits 1:112 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.

Return result (i.e., the value x in the binary floating-point working format).

bfp_CONVERT_FROM_SI64(x)
x is an integer value represented in signed doubleword integer format.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is equal to 0x0000_0000_0000_0000, result.class.Zero is set to 1.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exponent is set to the value 64.
Bits 0:64 of result.significand are set to the value of x sign-extended to 65 bits.

If bit 0 of result.significand is equal to 1, result.sign is set to 1, and result.significand is set to the value of the two's complement of result.significand.

If bit 0 of result.significand is equal to 0, result.significand is shifted left until bit 0 of result.significand is equal to 1, and result.exponent is decremented by the number of bits result.significand is shifted.

Return result (i.e., the value x in the binary floating-point working format).

bfp_CONVERT_FROM_SI128(x)
x is a 128-bit signed integer value.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is equal to 0x0000_0000_0000_0000_0000_0000_0000_0000,
result.class.Zero is set to 1.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to the contents of bit 0 of \(x\).
result.exponent is set to the value 128.
Bits 0:128 of result.significand are set to the value of \(x\) sign-extended to 129 bits.

If bit 0 of result.significand is equal to 1,
result.sign is set to 1, and
result.significand is set to the value of the two’s complement of result.significand.

If bit 0 of result.significand is equal to 0,
result.significand is shifted left until bit 0 of result.significand is equal to 1, and
result.exponent is decremented by the number of bits result.significand is shifted.

Return result (i.e., the value \(x\) in the binary floating-point working format).

\textbf{bfp\_CONVERT\_FROM\_UI64}(x)

\(x\) is an integer value represented in unsigned doubleword integer format.

Return \(x\) in the binary floating-point working format.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If \(x\) is equal to 0x0000_0000_0000_0000, do the following.
result.class.Zero is set to 1.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to 0.
result.exponent is set to the value 64.
Bits 0:64 of result.significand is set to the value of \(x\) zero-extended to 65 bits.

If bit 0 of result.significand is equal to 0, result.significand is shifted left until bit 0 of result.significand is equal to 1 and result.exponent is decremented by the number of bits result.significand is shifted.

Return result (i.e., the value \(x\) in the binary floating-point working format).

\textbf{bfp\_CONVERT\_FROM\_UI128}(x)

\(x\) is a 128-bit unsigned integer value.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If \( x \) is equal to 0x0000_0000_0000_0000_0000_0000_0000_0000, do the following.
result.class.Zero is set to 1.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to 0.
result.exponent is set to the value 128.
Bits 0:128 of result.significand are set to the value of \( x \) zero-extended to 129 bits.

If bit 0 of result.significand is equal to 0,
result.significand is shifted left until bit 0 of result.significand is equal to 1 and result.exponent is decremented by the number of bits result.significand is shifted.

Return result (i.e., the value \( x \) in the binary floating-point working format).

\[ \text{bfp\_DENORM}(x, y) \]
\( x \) is an integer value specifying the target format’s \( \text{Emin} \) value.
y is a binary floating-point value that is represented in the binary floating-point working format.

If \( y \).exponent is less than \( \text{Emin} \), let \( \text{sh\_cnt} \) be the value \( \text{Emin} - y \).exponent.
Otherwise, let \( \text{sh\_cnt} \) be the value 0.

\( y \).significand, having unbounded precision, is shifted right by \( \text{sh\_cnt} \) bits.
\( y \).exponent is incremented by \( \text{sh\_cnt} \).

Return \( y \) in the binary floating-point working format.

\[ \text{bfp\_DIVIDE}(x, y) \]
\( x \) is a binary floating-point value that is represented in the binary floating-point working format.
y is a binary floating-point value that is represented in the binary floating-point working format.

If \( x \) or \( y \) is an SNaN, \( \text{vxsnan\_flag} \) is set to 1.
Otherwise, if \( x \) and \( y \) are infinities, \( \text{vxidi\_flag} \) is set to 1.
Otherwise, if \( x \) and \( y \) are zeros, \( \text{vxzdz\_flag} \) is set to 1.
Otherwise, if \( x \) is a finite value and \( y \) is a zero, \( \text{zx\_flag} \) is set to 1.

If \( x \) is a QNaN, return \( x \).
Otherwise, if \( x \) is an SNaN, return \( x \) represented as a QNaN.
Otherwise, if \( y \) is a QNaN, return \( y \).
Otherwise, if \( y \) is an SNaN, return \( y \) represented as a QNaN.
Otherwise, if \( x \) and \( y \) are infinities, return the standard QNaN.
Otherwise, if \( x \) and \( y \) are zeros, return the standard QNaN.
Otherwise, if \( y \) is a zero, return infinity, having the sign of the exclusive-OR of the signs of \( x \) and \( y \).
Otherwise, return the normalized quotient of \( x \div y \), having unbounded range and precision.

\[ \text{bfp\_INFINITY} \]
The value +Infinity represented in the binary floating-point working format.

\[ \text{bfp\_INITIALIZE}(x) \]
Let \( x \).sign be set to 0.
Let \( x \).exponent be set to 0.
Let \( x \).significand be set to 0.
Let \( x \).class.SNaN be set to 0.
Let \( x \).class.QNaN be set to 0.
Let \( x \).class.Infinity be set to 0.
Let \( x \).class.Zero be set to 0.
Let \( x \).class.Denormal be set to 0.
Let $x\_c\_lass\_Normal$ be set to 0.

Return $x$.

$\textbf{bfp\_MULTIPLY}(x, y)$

$x$ is a binary floating-point value represented in the binary floating-point working format.
y is a binary floating-point value represented in the binary floating-point working format.

If $x$ or $y$ is an SNaN, $vx\_snan\_flag$ is set to 1.
Otherwise, if $x$ is an infinity and $y$ is a zero, $vx\_imz\_flag$ is set to 1.
Otherwise, if $x$ is a zero and $y$ is an infinity, $vx\_imz\_flag$ is set to 1.

If $x$ is a QNaN, return $x$.
Otherwise, if $x$ is an SNaN, return $x$ represented as a QNaN.
Otherwise, if $y$ is a QNaN, return $y$.
Otherwise, if $y$ is an SNaN, return $y$ represented as a QNaN.
Otherwise, if $x$ is an infinity and $y$ is a zero, return the standard QNaN.
Otherwise, if $x$ is a zero and $y$ is an infinity, return the standard QNaN.
Otherwise, return the normalized product of $x \times y$, having unbounded range and precision.

$\textbf{bfp\_MULTIPLY\_ADD}(x, y, z)$

$x$ is a binary floating-point value represented in the binary floating-point working format.
y is a binary floating-point value represented in the binary floating-point working format.
z is a binary floating-point value represented in the binary floating-point working format.

If $x$, $y$, or $z$ is an SNaN, $vx\_snan\_flag$ is set to 1.
Otherwise, if $x$ is an infinity and $y$ is a zero, $vx\_imz\_flag$ is set to 1.
Otherwise, if $x$ is a zero and $y$ is an infinity, $vx\_imz\_flag$ is set to 1.

Otherwise, if $z$ and the product of $x \times y$ are Infinity values having opposite signs, $vx\_isi\_flag$ is set to 1.

If $x$ is a QNaN, return $x$.
Otherwise, if $x$ is an SNaN, return $x$ represented as a QNaN.
Otherwise, if $z$ is a QNaN, return $z$.
Otherwise, if $z$ is an SNaN, return $z$ represented as a QNaN.
Otherwise, if $x$ is an infinity and $y$ is a zero, return the standard QNaN.
Otherwise, if $x$ is a zero and $y$ is an infinity, return the standard QNaN.
Otherwise, if $z$ and the product of $x \times y$ are Infinity values having opposite signs, return the standard QNaN.
Otherwise, return the sum of $z$ and the normalized product of $x \times y$, having unbounded range and precision.

$\textbf{bfp\_NEGATE}(x)$

$x$ is a binary floating-point value that is represented in the binary floating-point working format.

If $x$ is not a NaN, return $x$ with its sign complemented. Otherwise, return $x$.

$\textbf{bfp\_NMAX\_BFLOAT16}$

Return the largest positive normalized bfloat16 floating-point value (i.e., $2^{128}.2^{128-1}$) represented in the binary floating-point working format.

return bfp\_CONVERT\_FROM\_BFLOAT16(0x7F7F)

$\textbf{bfp\_NMAX\_BFLOAT16}$

Return the largest positive normalized half-precision floating-point value (i.e., $2^{16}.2^{16-11}$), represented in the binary floating-point working format.

return bfp\_CONVERT\_FROM\_BFLOAT16(0x7BFF)
bfp_NMAX_BFP64
Return the largest finite double-precision floating-point value (i.e., $2^{1024-53}$) in the binary floating-point working format.

return bfp_CONVERT_FROM_BFP64(0x7FEF_FFFF_FFFF_FFFF)

bfp_NMAX_BFP80
Return the largest finite double-extended-precision floating-point value (i.e., $2^{16384-65}$) in the binary floating-point working format.

return bfp_CONVERT_FROM_BFP80(0x7FFE_FFFF_FFFF_FFFF_FFFF)

bfp_NMAX_BFP128
Return the largest finite quad-precision value (i.e., $2^{16384-113}$) in the binary floating-point working format.

return bfp_CONVERT_FROM_BFP128(0x7FFE_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF)

bfp_NMIN_BFLOAT16
Return the smallest positive normalized bfloat16 floating-point value (i.e., $2^{-126}$), represented in the binary floating-point working format.

return bfp_CONVERT_FROM_BFLOAT16(0x0080)

bfp_NMIN_BFP16
Return the smallest positive normalized half-precision floating-point value, $2^{-14}$, represented in the binary floating-point working format.

return bfp_CONVERT_FROM_BFP16(0x0400)

bfp_NMIN_BFP64
Return the smallest positive normalized double-precision floating-point value, $2^{-1022}$, represented in the binary floating-point working format.

return bfp_CONVERT_FROM_BFP64(0x0010_0000_0000_0000)

bfp_NMIN_BFP80
Return the smallest positive normalized double-extended-precision floating-point value, $2^{-16382}$, represented in the binary floating-point working format.

return bfp_CONVERT_FROM_BFP80(0x0001_0000_0000_0000_0000)

bfp_NMIN_BFP128
Return the smallest, positive, normalized quad-precision floating-point value, $2^{-16382}$, represented in the binary floating-point working format.

return bfp_CONVERT_FROM_BFP128(0x0001_0000_0000_0000_0000_0000_0000_0000)

bfp_QUIET(x)
\(x\) is a Signalling NaN.

Return \(x\) converted to a Quiet NaN with \(x.class.QNaN\) set to 1 and \(x.class.SNaN\) set to 0.

bfp_ROUND_CEIL(p, x)
\(x\) is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision. \(x\) must be rounded as presented, without prenormalization.

\(p\) is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return the smallest floating-point number having unbounded exponent range and a significand with a width of \(p\) bits that is greater or equal in value to \(x\).
inc_flag is set to 1 if the magnitude of the value returned is greater than \( x \).
xx_flag is set to 1 if the value returned is not equal to \( x \).

**bfp_ROUND_FLOOR(p, x)**

\( x \) is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision. The value must be rounded as presented, without prenormalization.

\( p \) is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return the largest floating-point number having unbounded exponent range and a significand with a width of \( p \) bits that is lesser or equal in value to \( x \).

inc_flag is set to 1 if the magnitude of the value returned is greater than \( x \).
xx_flag is set to 1 if the value returned is not equal to \( x \).

**bfp_ROUND_TO_BFLOAT16_NO_TRAP(x)**

\( x \) is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

Return the value \( x \), rounded to bfloat16 significand precision and exponent range under control of the rounding mode specified in RN, and represented in the binary floating-point working format.

With respect to this rounding function, OE and UE are ignored (treated as if OE=0 and UE=0).

```
rmode ← FPSCR.RN
if x.class.SNaN then do
  vxsnan_flag ← 1
  return bfp_QUIET(x)
end
if x.class.QNaN then return x
if x.class.Infinity then return x
if x.class.Zero then return x
if bfp_ABSOLUTE(x)<bfp_NMIN_BFLOAT16 then do
  x ← bfp_DENORM(-126,x)
  if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(8,x)
  if rmode=0b01 then r ← bfp_ROUND_TRUNC(8,x)
  if rmode=0b10 then r ← bfp_ROUND_CEIL(8,x)
  if rmode=0b11 then r ← bfp_ROUND_FLOOR(8,x)
  ux_flag ← xx_flag
  return r
end
if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(8,x)
if rmode=0b01 then r ← bfp_ROUND_TRUNC(8,x)
if rmode=0b10 then r ← bfp_ROUND_CEIL(8,x)
if rmode=0b11 then r ← bfp_ROUND_FLOOR(8,x)
if bfp_ABSOLUTE(r)>bfp_NMAX_BFLOAT16 then do
  if rmode=0b00 then r ← x.sign ? bfp_INFINITY : bfp_INFINITY
  if rmode=0b01 then r ← x.sign ? bfp_NMAX_BFLOAT16 : bfp_NMAX_BFLOAT16
  if rmode=0b10 then r ← x.sign ? bfp_NMAX_BFLOAT16 : bfp_INFINITY
  if rmode=0b11 then r ← x.sign ? bfp_INFINITY : bfp_NMAX_BFLOAT16
```
r.sign ← x.sign

ox_flag ← 0b1
xx_flag ← 0b1
inf_flag ← 0bU
end

return r

\texttt{bfp\_ROUND\_TO\_BFP16}(x, y)

\(y\) is a normalized floating-point value represented in the binary floating-point working format, having unbounded exponent range and significand precision.

\(x\) is a 2-bit integer value specifying one of four rounding modes.

0b00 Round to Nearest Even
0b01 Round towards Zero
0b10 Round towards +Infinity
0b11 Round towards -Infinity

If \(y\) is an QNaN, Infinity, or Zero, return \(y\). Otherwise, if \(y\) is an SNaN, set \(\text{vxsnan\_flag}\) to 1 and return the corresponding QNaN representation of \(y\). Otherwise, return the value \(y\) rounded to half-precision format's exponent range and significand precision using the rounding mode specified by \(x\).

if \text{y.class.Zero} \text{ or } \text{y.class.Infinity} \text{ then return}(y)
if \text{y.class.QNaN} \text{ or } \text{y.class.SNaN} \text{ then do}
result ← \(y\)
result.significand.bit[1] ← 1
result.significand.bit[11:inf] ← 0
result.class.SNaN ← 0
result.class.QNaN ← 1
\(\text{vxsnan\_flag} ← \text{y.class.SNaN}\)
return(result)
end

if \text{bfp\_COMPARE\_LT}(y, \text{bfp\_NMIN\_BFP16}) \text{ then do}
if FPSCR.UE=0 then do
do while \(y\).exponent < -14 \text{ // denormalize } y
\(y\).significand ← \(y\).significand >> 1
\(y\).exponent ← \(y\).exponent + 1
end
if x=0b00 then result ← \text{bfp\_ROUND\_TO\_BFP16\_NEAR\_EVEN}(y)
if x=0b01 then result ← \text{bfp\_ROUND\_TO\_BFP16\_TRUNC}(y)
if x=0b10 then result ← \text{bfp\_ROUND\_TO\_BFP16\_CEIL}(y)
if x=0b11 then result ← \text{bfp\_ROUND\_TO\_BFP16\_FLOOR}(y)
do while result.significand.bit[0] = 0 \text{ // normalize result}
result.significand ← result.significand << 1
result.exponent ← result.exponent - 1
end
ux_flag ← xx_flag
return(result)
end
else do
\(y\).exponent ← \(y\).exponent + 24
ux_flag ← 1
end
end

if x=0b00 then result ← \text{bfp\_ROUND\_TO\_BFP16\_NEAR\_EVEN}(y)
if $x=0b01$ then $\text{result} \leftarrow \text{bfp\_ROUND\_TO\_BFP16\_TRUNC}(y)$
if $x=0b10$ then $\text{result} \leftarrow \text{bfp\_ROUND\_TO\_BFP16\_CEIL}(y)$
if $x=0b11$ then $\text{result} \leftarrow \text{bfp\_ROUND\_TO\_BFP16\_FLOOR}(y)$

if bfp\_COMPARE\_GT($\text{result}$, bfp\_NMAX\_BFP16) then do
  if $\text{OE}=0$ then do
    if $x=0b00$ then $\text{result} \leftarrow \text{sign} ? \text{bfp\_NEGATE}(\text{bfp\_INFINITY}) : \text{bfp\_INFINITY}$
    if $x=0b01$ then $\text{result} \leftarrow \text{sign} ? \text{bfp\_NEGATE}(\text{bfp\_NMAX\_BFP16}) : \text{bfp\_NMAX\_BFP16}$
    if $x=0b10$ then $\text{result} \leftarrow \text{sign} ? \text{bfp\_NEGATE}(\text{bfp\_NMAX\_BFP16}) : \text{bfp\_INFINITY}$
    if $x=0b11$ then $\text{result} \leftarrow \text{sign} ? \text{bfp\_NEGATE}(\text{bfp\_INFINITY}) : \text{bfp\_NMAX\_BFP16}$
  ox\_flag ← 0b1
  xx\_flag ← 0b1
  inc\_flag ← 0bU
  return($\text{result}$)
else do
  $\text{result}.\text{exponent} \leftarrow \text{result}.\text{exponent} - 24$
  ox\_flag ← 1
end
end
return($\text{result}$)

\text{bfp\_ROUND\_TO\_BFP16\_CEIL}(x)

$x$ is a normalized floating-point value represented in the binary floating-point working format, having unbounded exponent range and significand precision.

Return the smallest floating-point number having unbounded exponent range but half-precision significand precision that is greater or equal in value to $x$.

If the magnitude of the value returned is greater than $x$, inc\_flag is set to 1.

If the value returned is not equal to $x$, xx\_flag is set to 1.

\text{bfp\_ROUND\_TO\_BFP16\_FLOOR}(x)

$x$ is a normalized floating-point value represented in the binary floating-point working format, having unbounded exponent range and significand precision.

Return the largest floating-point number having unbounded exponent range but half-precision significand precision that is lesser or equal in value to $x$.

If the magnitude of the value returned is greater than $x$, inc\_flag is set to 1.

If the value returned is not equal to $x$, xx\_flag is set to 1.

\text{bfp\_ROUND\_TO\_BFP16\_NEAR\_EVEN}(x)

$x$ is a normalized floating-point value represented in the binary floating-point working format, having unbounded exponent range and significand precision.

Return the floating-point number having unbounded exponent range but half-precision significand precision that is nearest in value to $x$ (in case of a tie, the floating-point number having unbounded exponent range but half-precision significand precision with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than $x$, inc\_flag is set to 1.

If the value returned is not equal to $x$, xx\_flag is set to 1.

\text{bfp\_ROUND\_TO\_BFP16\_TRUNC}(x)

$x$ is a normalized floating-point value represented in the binary floating-point working format, having unbounded exponent range and significand precision.
Return the largest floating-point number having unbounded exponent range but half-precision significand precision that is lesser or equal in value to $x$ if $x > 0$, or the smallest floating-point number having unbounded exponent range but half-p-precision significand precision that is greater or equal in value to $x$ if $x < 0$.

If the magnitude of the value returned is greater than $x$, `inc_flag` is set to 1.

If the value returned is not equal to $x$, `xx_flag` is set to 1.

```plaintext
bfp_ROUND_TO_BFP32_SIGNIFICAND(x)
```

$x$ is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

Return the value $x$ rounded to 24-bit significand precision under control of the rounding mode specified in $RN$, retaining unbounded exponent range, represented in the binary floating-point working format.

```plaintext
rmode ← FPSCR.RN
if x.class.QNaN | x.class.Infinity | x.class.Zero then return x
if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(24, x)
if rmode=0b01 then r ← bfp_ROUND_TRUNC(24, x)
if rmode=0b10 then r ← bfp_ROUND_CEIL(24, x)
if rmode=0b11 then r ← bfp_ROUND_FLOOR(24, x)
return r (binary floating-point working format)
```

```plaintext
bfp_ROUND_TO_BFP32_NO_TRAP(x)
```

$x$ is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

Return the value $x$ rounded to single-precision under control of the rounding mode specified in $RN$, represented in double-precision format.

Note that $OE$ and $UE$ are ignored, results are returned as if $OE=0$ and $UE=0$.

```plaintext
rmode = FPSCR.RN
if x.class.QNaN | x.class.Infinity | x.class.Zero then return x
if bfp_ABSOLUTE(x)<bfp_NMIN_BFP32 then do
    x = bfp_DENORM(-126, x)
    if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(24, x)
    if rmode=0b01 then r ← bfp_ROUND_TRUNC(24, x)
    if rmode=0b10 then r ← bfp_ROUND_CEIL(24, x)
    if rmode=0b11 then r ← bfp_ROUND_FLOOR(24, x)
    ux_flag = xx_flag
    return r
end
if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(24, x)
if rmode=0b01 then r ← bfp_ROUND_TRUNC(24, x)
if rmode=0b10 then r ← bfp_ROUND_CEIL(24, x)
if rmode=0b11 then r ← bfp_ROUND_FLOOR(24, x)
if bfp_ABSOLUTE(r)>bfp_NMAX_BFP32 then do
    if rmode=0b00 then r = x.sign ? bfp_INFINITY   : bfp_INFINITY
    if rmode=0b01 then r = x.sign ? bfp_NMAX_BFP32 : bfp_NMAX_BFP32
    if rmode=0b10 then r = x.sign ? bfp_NMAX_BFP32 : bfp_INFINITY
    if rmode=0b11 then r = x.sign ? bfp_INFINITY   : bfp_NMAX_BFP32
    r.sign   = x.sign
```
```
ox_flag  = 0b1
xx_flag  = 0b1
inc_flag = 0bU
end
return r
```

(binary floating-point working format)

**bfp_ROUND_TO_BFP64(ro, rmode, x)**

- **x** is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.
- **ro** is a 1-bit unsigned integer and **rmode** is a 2-bit unsigned integer, together specifying one of five rounding modes to be used in rounding *z*.

```
ro=0 rmode=0b00  Round to Nearest Even
ro=0 rmode=0b01  Round towards Zero
ro=0 rmode=0b10  Round towards +Infinity
ro=0 rmode=0b11  Round towards -Infinity
ro=1              Round to Odd
```

Return the value *x* rounded to double-precision under control of the specified rounding mode.

```
if x.class.QNaN     then return x
if x.class.Infinity then return x
if x.class.Zero     then return x
if bfp_ABSOLUTE(x)<bfp_NMIN_BFP64  then do
  if FPSCR.UE=0 then do
    x ← bfp_DENORM(-1022,x)
    if ro=0 & rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(53,x)
    if ro=0 & rmode=0b01 then r ← bfp_ROUND_TRUNC(53,x)
    if ro=0 & rmode=0b10 then r ← bfp_ROUND_CEIL(53,x)
    if ro=0 & rmode=0b11 then r ← bfp_ROUND_FLOOR(53,x)
    if ro=1              then r ← bfp_ROUND_ODD(53,x)
    ux_flag ← xx_flag
    return(r)
  end
else do
  x.exponent ← x.exponent + 1536
  ux_flag ← 1
end
```

```
if ro=0 & rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(53,x)
if ro=0 & rmode=0b01 then r ← bfp_ROUND_TRUNC(53,x)
if ro=0 & rmode=0b10 then r ← bfp_ROUND_CEIL(53,x)
if ro=0 & rmode=0b11 then r ← bfp_ROUND_FLOOR(53,x)
if ro=1              then r ← bfp_ROUND_ODD(53,x)
if bfp_ABSOLUTE(r)>bfp_NMAX_BFP64 then do
  if FPSCR.OE=0 then do
    if ro=0 & rmode=0b00 then r ← x.sign ? bfp_INFINITY   : bfp_INFINITY
    if ro=0 & rmode=0b01 then r ← x.sign ? bfp_NMAX_BFP64 : bfp_NMAX_BFP64
    if ro=0 & rmode=0b10 then r ← x.sign ? bfp_NMAX_BFP64 : bfp_INFINITY
    if ro=0 & rmode=0b11 then r ← x.sign ? bfp_INFINITY   : bfp_NMAX_BFP64
    if ro=1              then r ← x.sign ? bfp_NMAX_BFP64 : bfp_NMAX_BFP64
    r.sign ← x.sign
    ox_flag ← 0b1
    xx_flag ← 0b1
    inc_flag ← 0bU
    return(r)
  end
else do
```
bfp_ROUND_TO_BFP64_NO_TRAP(x)

$x$ is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

Return the value $x$ rounded to double-precision under control of the rounding mode specified in RN, represented in double-precision format.

Note that OE and UE are ignored, results are returned as if OE=0 and UE=0.

```c
rmode = FPSCR.RN
if x.class.QNaN | x.class.Infinity | x.class.Zero then return x
if bfp_ABSOLUTE(x)<bfp_NMIN_BFP64 then do
  x = bfp_DENORM(-1022,x)
  if rmode=0b00 then r = bfp_ROUND_NEAR_EVEN(53,x)
  if rmode=0b01 then r = bfp_ROUND_TRUNC(53,x)
  if rmode=0b10 then r = bfp_ROUND_CEIL(53,x)
  if rmode=0b11 then r = bfp_ROUND_FLOOR(53,x)
  ox_flag = xx_flag
  return r
end
if rmode=0b00 then r = bfp_ROUND_NEAR_EVEN(53,x)
if rmode=0b01 then r = bfp_ROUND_TRUNC(53,x)
if rmode=0b10 then r = bfp_ROUND_CEIL(53,x)
if rmode=0b11 then r = bfp_ROUND_FLOOR(53,x)
if bfp_ABSOLUTE(r)>bfp_NMAX_BFP64 then do
  if rmode=0b00 then r = x.sign ? bfp_INFINITY   : bfp_INFINITY
  if rmode=0b01 then r = x.sign ? bfp_NMAX_BFP64 : bfp_NMAX_BFP64
  if rmode=0b10 then r = x.sign ? bfp_NMAX_BFP64 : bfp_INFINITY
  if rmode=0b11 then r = x.sign ? bfp_INFINITY   : bfp_NMAX_BFP64
  r.sign   = x.sign
  ox_flag  = 0b1
  xx_flag  = 0b1
  inc_flag = 0bU
end
return r (binary floating-point working format)
```

bfp_ROUND_TO_BFP80(rmode,x)

$x$ is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

$rmode$ is a 2-bit unsigned integer, together specifying one of four rounding modes to be used in rounding $x$.

- $rmode=0b00$ Round to Nearest Even
- $rmode=0b01$ Round towards Zero
- $rmode=0b10$ Round towards $+\infty$
- $rmode=0b11$ Round towards $-\infty$

Return the value $x$ rounded to double-extended-precision under control of the specified rounding mode.

```c
if x.class.QNaN then return x
```

596  Power ISA™ I
if \( x.\text{class} = \text{Infinity} \) then return \( x \)
if \( x.\text{class} = \text{Zero} \) then return \( x \)
if \( \text{bfp\_ABSOLUTE}(x) < \text{bfp\_NMN\_BFP80} \) then do
  if FPSCR.UE = 0 then do
    \( x \leftarrow \text{bfp\_DENORM(-16382, x)} \)
    if rmode = 0b00 then \( r \leftarrow \text{bfp\_ROUND\_NEAR\_EVEN}(64, x) \)
    if rmode = 0b01 then \( r \leftarrow \text{bfp\_ROUND\_TRUNC}(64, x) \)
    if rmode = 0b10 then \( r \leftarrow \text{bfp\_ROUND\_CEIL}(64, x) \)
    if rmode = 0b11 then \( r \leftarrow \text{bfp\_ROUND\_FLOOR}(64, x) \)
    \( ux\_flag \leftarrow xx\_flag \)
    return(\( r \))
  end
else do
  \( x.\text{exponent} \leftarrow x.\text{exponent} + 24576 \)
  \( ux\_flag \leftarrow 1 \)
end
if rmode = 0b00 then \( r \leftarrow \text{bfp\_ROUND\_NEAR\_EVEN}(64, x) \)
if rmode = 0b01 then \( r \leftarrow \text{bfp\_ROUND\_TRUNC}(64, x) \)
if rmode = 0b10 then \( r \leftarrow \text{bfp\_ROUND\_CEIL}(64, x) \)
if rmode = 0b11 then \( r \leftarrow \text{bfp\_ROUND\_FLOOR}(64, x) \)
if \( \text{bfp\_ABSOLUTE}(r) > \text{bfp\_NMAX\_BFP80} \) then do
  if FPSCR.OE = 0 then do
    if rmode = 0b00 then \( r \leftarrow x.\text{sign} ? \text{bfp\_INFINITY} : \text{bfp\_INFINITY} \)
    if rmode = 0b01 then \( r \leftarrow x.\text{sign} ? \text{bfp\_NMAX\_BFP80} : \text{bfp\_NMAX\_BFP80} \)
    if rmode = 0b10 then \( r \leftarrow x.\text{sign} ? \text{bfp\_NMAX\_BFP80} : \text{bfp\_INFINITY} \)
    if rmode = 0b11 then \( r \leftarrow x.\text{sign} ? \text{bfp\_INFINITY} : \text{bfp\_NMAX\_BFP80} \)
  end
  \( r.\text{sign} \leftarrow x.\text{sign} \)
  \( ox\_flag \leftarrow 0b1 \)
  \( xx\_flag \leftarrow 0b1 \)
  \( inc\_flag \leftarrow 0bU \)
  return(\( r \))
end
else do
  \( r.\text{exponent} \leftarrow r.\text{exponent} - 24576 \)
  \( ox\_flag \leftarrow 1 \)
end
return \( r \) (binary floating-point working format)

\text{bfp\_ROUND\_TO\_BFP128}(ro, rmode, x)

\( x \) is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

\( ro \) is a 1-bit unsigned integer and \( rmode \) is a 2-bit unsigned integer, together specifying one of five rounding modes to be used in rounding \( z \).

\begin{align*}
  ro &= 0 & rmode &= 0b00 & \text{Round to Nearest Even} \\
  ro &= 0 & rmode &= 0b01 & \text{Round towards Zero} \\
  ro &= 0 & rmode &= 0b10 & \text{Round towards +Infinity} \\
  ro &= 0 & rmode &= 0b11 & \text{Round towards -Infinity} \\
  ro &= 1 & \text{Round to Odd}
\end{align*}

Return the value \( x \) rounded to quad-precision under control of the specified rounding mode.

if \( x.\text{class} = \text{QNaN} \) then return \( x \)
if \( x.\text{class} = \text{Infinity} \) then return \( x \)
if \( x.\text{class} = \text{Zero} \) then return \( x \)
if \( \text{bfp\_ABSOLUTE}(x) < \text{bfp\_NMN\_BFP128} \) then do
if FPSCR.UE=0 then do
  x ← bfp_DENORM(16382,x)
if ro=0 & rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(113,x)
if ro=0 & rmode=0b01 then r ← bfp_ROUND_TRUNC(113,x)
if ro=0 & rmode=0b10 then r ← bfp_ROUND_CEIL(113,x)
if ro=0 & rmode=0b11 then r ← bfp_ROUND_FLOOR(113,x)
if ro=1 then r ← bfp_ROUND_ODD(113,x)
ux_flag ← xx_flag
return(r)
end
else do
  x.exponent ← x.exponent + 24576
  ux_flag ← 1
end

if ro=0 & rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(113,x)
if ro=0 & rmode=0b01 then r ← bfp_ROUND_TRUNC(113,x)
if ro=0 & rmode=0b10 then r ← bfp_ROUND_CEIL(113,x)
if ro=0 & rmode=0b11 then r ← bfp_ROUND_FLOOR(113,x)
if ro=1 then r ← bfp_ROUND_ODD(113,x)
if bfp_ABSOLUTE(r)>bfp_NMAX_BFP128 then do
  if FPSCR.OE=0 then do
    if ro=0 & rmode=0b00 then r ← bfp_INFINITY
    if ro=0 & rmode=0b01 then r ← bfp_NMAX_BFP128
    if ro=0 & rmode=0b10 then r ← bfp_NMAX_BFP128
    if ro=0 & rmode=0b11 then r ← bfp_INFINITY
    if ro=1 then r ← bfp_NMAX_BFP128
  r.sign ← x.sign
  ox_flag ← 0b1
  xx_flag ← 0b1
  inc_flag ← 0bU
  return(r)
  end
else do
  r.exponent ← r.exponent - 24576
  ox_flag ← 1
end
end

return r  (binary floating-point working format)
return the largest double-precision floating-point integer value that is lesser or equal in value to $x$ if $x>0$, or the smallest double-precision floating-point integer value that is greater or equal in value to $x$ if $x<0$.

- If $rmode=0b010$ (Round towards $+\infty$), return the smallest double-precision floating-point integer value that is greater or equal in value to $x$.
- If $rmode=0b011$ (Round towards $-\infty$), return the largest double-precision floating-point integer value that is lesser or equal in value to $x$.
- If $rmode=0b100$ (Round to Nearest Away), return the double-precision floating-point integer value that is nearest in value to $x$ (in case of a tie, the double-precision floating-point integer value that is furthest away from 0 is used).

`inc_flag` is set to 1 if the magnitude of the value returned is greater than $x$.
`xx_flag` is set to 1 if the value returned is not equal to $x$.

**bfp_ROUND_ODD(p, x)**

$x$ is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision. $x$ must be rounded as presented, without prenormalization.

$p$ is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return $x$ with bit $p-1$ of the significand set to 1 if any of the bits to the right of bit $p-1$ of the significand of $x$ are equal to 1, and all bits to the right of bit $p-1$ of the significand of the value returned are set to 0. Otherwise return $x$ with all bits to the right of bit $p-1$ of the significand set to 0.

`inc_flag` is set to 1 if the magnitude of the value returned is greater than $x$.
`xx_flag` is set to 1 if the value returned is not equal to $x$.

**bfp_ROUND_NEAR_EVEN(p, x)**

$x$ is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision. $x$ must be rounded as presented, without prenormalization.

$p$ is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return the floating-point number having unbounded exponent range and a significand with a width of $p$ bits that is nearest in value to $x$ (in case of a tie, the floating-point number having unbounded exponent range and a $p$-bit significand with the least-significant bit equal to 0 is used).

`inc_flag` is set to 1 if the magnitude of the value returned is greater than $x$.
`xx_flag` is set to 1 if the value returned is not equal to $x$.

**bfp_ROUND_TRUNC(p, x)**

$x$ is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision. $x$ must be rounded as presented, without prenormalization.

$p$ is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return the largest floating-point number having unbounded exponent range and a significand with a width of $p$ bits that is lesser or equal in value to $x$ if $x>0$, or the smallest floating-point number having unbounded exponent range but double-precision significand precision that is greater or equal in value to $x$ if $x<0$.

`inc_flag` is set to 1 if the magnitude of the value returned is greater than $x$.
`xx_flag` is set to 1 if the value returned is not equal to $x$. 
**bfp_SQUARE_ROOT(x)**

x is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

If x is an SNaN, vxsnan_flag is set to 1.
Otherwise, if x is negative and non-zero, vxsqrt_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is -Zero, return -Zero.
Otherwise, if x is negative, return the standard QNaN.
Otherwise, return the normalized square root of x, represented in the binary floating-point working format, having unbounded range and precision.

**bfp16_CONVERT_FROM_BFP(x)**

x is a floating-point value represented in the binary floating-point working format.

If x.class.QNaN=1, do the following.
  Bit 0 of result is set to the value of x.sign.
  Bits 1:5 of result are set to the value 0b11111.
  Bits 6:15 of result are set to the value of bits 1:10 of x.significand.

Otherwise, if x.class.Infinity=1, do the following.
  Bit 0 of result is set to the value of x.sign.
  Bits 1:5 of result are set to the value 0b11111.
  Bits 6:15 of result are set to 0.

Otherwise, if x.class.Zero=1, do the following.
  Bit 0 of result is set to the value of x.sign.
  Bits 1:15 of result are set to 0.

Otherwise, if x.exponent is less than -14 and UE=0, do the following.
  Bit 0 of result is set to the value of x.sign.
  sh_cnt is set to the difference, -14 - x.exponent.
  Bits 1:5 of result are set to 0b00000.
  Bits 6:15 of result are set to bits 1:10 of x.significand shifted right by sh_cnt bits.

Otherwise, if x.exponent is less than -14 and UE=1, result is undefined.
Otherwise, if x.exponent is greater than 15 and OE=1, result is undefined.

Otherwise, do the following.
  Bit 0 of result is set to the value of x.sign.
  Bits 1:5 of result are set to the sum, x.exponent + 15.
  Bits 6:15 of result are set to bits 1:10 of x.significand.

Return result.

**bfp32_ABSOLUTE(x)**

x is a floating-point value represented in single-precision format.

Return x with its sign set to 0.

**bfp32_CONVERT_FROM_BFP(x)**

x is a floating-point value represented in the binary floating-point working format.

If x.class.QNaN=1, do the following.
  Bit 0 of result is set to the value of x.sign.
  Bits 1:8 of result are set to the value 0b1111_1111.
  Bits 9:31 of result are set to the value of bits 1:23 of x.significand.
Otherwise, if $x.class.Infinity=1$, do the following.
- Bit 0 of $\text{result}$ is set to the value of $x.sign$.
- Bits 1:9 of $\text{result}$ are set to the value $0b1111_1111$.
- Bits 9:31 of $\text{result}$ are set to 0.

Otherwise, if $x.class.Zero=1$, do the following.
- Bit 0 of $\text{result}$ is set to the value of $x.sign$.
- Bits 1:31 of $\text{result}$ are set to 0.

Otherwise, if $x.exponent$ is less than $-126$ and $UE=0$, do the following.
- Bit 0 of $\text{result}$ is set to the value of $x.sign$.
- $\text{sh_cnt}$ is set to the difference, $-126 - x.exponent$.
- Bits 1:8 of $\text{result}$ are set to $0b0000_0000$.
- Bits 9:31 of $\text{result}$ are set to bits 1:23 of $x.significand$ shifted right by $\text{sh_cnt}$ bits.

Otherwise, if $x.exponent$ is less than $-126$ and $UE=1$, $\text{result}$ is undefined.

Otherwise, if $x.exponent$ is greater than $127$ and $OE=1$, $\text{result}$ is undefined.

Otherwise, do the following.
- Bit 0 of $\text{result}$ is set to the value of $x.sign$.
- Bits 1:8 of $\text{result}$ are set to the sum, $x.exponent + 127$.
- Bits 9:31 of $\text{result}$ are set to bits 1:23 of $x.significand$.

Return $\text{result}$.

### bfp32\_CONVERT\_FROM\_BFP64$(x)$

$x$ is a single-precision floating-point value in double-precision format.

Returns the value $x$ in single-precision format. $x$ must be representable in single-precision, or else $\text{result}$ returned is undefined. $x$ may require denormalization. No rounding is performed. If $x$ is a SNaN, it is converted to a single-precision SNaN having the same payload as $x$.

$\text{sign} \leftarrow x.\text{bit}[0]$
$\text{exp} \leftarrow x.\text{bit}[1:11] - 1023$
$\text{frac} \leftarrow x.\text{bit}[12:63]$

if $(\text{exp} = -1023) \& (\text{frac} = 0) \& (\text{sign}=0)$ then return($0x0000_0000$) // +Zero
else if $(\text{exp} = -1023) \& (\text{frac} = 0) \& (\text{sign}=1)$ then return($0x8000_0000$) // -Zero
else if $(\text{exp} = -1023) \& (\text{frac} != 0)$ then return($0xUUUU_UUUU$) // DP denorm
else if $(\text{exp} < -126)$ then do // denormalization required
    \text{msb} = 1
    do while $(\text{exp} < -126)$ // denormalize operand until exp=Emin
        $\text{frac}.\text{bit}[1:51] \leftarrow \text{frac}.\text{bit}[0:50]$
        $\text{frac}.\text{bit}[0] \leftarrow \text{msb}$
        \text{msb} \leftarrow 0
        \text{exp} \leftarrow \text{exp} + 1
    end
    if $(\text{frac} = 0)$ then return($0xUUUU_UUUU$) // value not representable in SP format
else do // return denormal SP
    $\text{result}.\text{bit}[0] \leftarrow \text{sign}$
    $\text{result}.\text{bit}[1:8] \leftarrow 0$
    $\text{result}.\text{bit}[9:31] \leftarrow \text{frac}.\text{bit}[0:22]$
    return($\text{result}$)
end
end
else if $(\text{exp} = +1024) \& (\text{frac} = 0) \& (\text{sign}=0)$ then return($0x7F80_0000$) // +Infinity
else if $(\text{exp} = +1024) \& (\text{frac} = 0) \& (\text{sign}=1)$ then return($0xFF80_0000$) // -Infinity
else if $(\text{exp} = +1024) \& (\text{frac} != 0)$ then do // QNaN or SNaN
    $\text{result}.\text{bit}[0] \leftarrow \text{sign}$
    $\text{result}.\text{bit}[1:8] \leftarrow 255$
result.bit[9:31] ← frac.bit[0:22]
return(result)
end
else if (exp < +1024) & (exp > +126) then return(0xUUUU_UUUU) // overflow
else do // normal value
    result.bit[0] ← sign
    result.bit[9:31] ← frac.bit[0:22]
    return(result)
end

bfp32_MAXIMUM(x, y)
x is a binary floating-point value that is represented in single-precision format.
y is a binary floating-point value that is represented in single-precision format.

If x or y is an SNaN, vxsnan_flag is set to 1.
If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the greater of x and y, where +0 is considered greater than -0.

bfp32_MINIMUM(x, y)
x is a binary floating-point value that is represented in single-precision format.
y is a binary floating-point value that is represented in single-precision format.

If x or y is an SNaN, vxsnan_flag is set to 1.
If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the lesser of x and y, where -0 is considered less than +0.

bfp32_NEGATE(x)
x is a floating-point value represented in single-precision format.

Return x with its sign complemented.

bfp32_NEGATIVE_ABSOLUTE(x)
x is a floating-point value represented in single-precision format.

Return x with its sign set to 1.

bfp64_ABSOLUTE(x)
x is a floating-point value represented in double-precision format.

Return x with its sign set to 0.

bfp64_CONVERT_FROM_BFP(x)
x is a floating-point value represented in the binary floating-point working format.

If x.class.QNaN=1, do the following.
    Bit 0 of result is set to the value of x.sign.
    Bits 1:11 of result are set to the value 0b111_1111_1111.
    Bits 12:63 of result are set to the value of bits 1:52 of x.significand.
Otherwise, if \(x.\text{class}.\text{Infinity}=1\), do the following.

- Bit 0 of result is set to the value of \(x.\text{sign}\).
- Bits 1:11 of result are set to the value \(0\text{b}11_111_1111\).
- Bits 12:63 of result are set to 0.

Otherwise, if \(x.\text{class}.\text{Zero}=1\), do the following.

- Bit 0 of result is set to the value of \(x.\text{sign}\).
- Bits 1:63 of result are set to 0.

Otherwise, if \(x.\text{exponent}\) is less than \(-1022\) and \(UE=0\), do the following.

- Bit 0 of result is set to the value of \(x.\text{sign}\).
- \(sh\_cnt\) is set to the difference, \(-1022 - x.\text{exponent}\).
- Bits 1:11 of result are set to \(0\text{b}000\_0000\_0000\).
- Bits 12:63 of result are set to bits 1:52 of \(x.\text{significand}\) shifted right by \(sh\_cnt\) bits.

Otherwise, if \(x.\text{exponent}\) is less than \(-1022\) and \(UE=1\), result is undefined.

Otherwise, if \(x.\text{exponent}\) is greater than \(1023\) and \(OE=1\), result is undefined.

Otherwise, do the following.

- Bit 0 of result is set to the value of \(x.\text{sign}\).
- Bits 1:11 of result are set to the sum, \(x.\text{exponent} + 1023\).
- Bits 12:63 of result are set to bits 1:52 of \(x.\text{significand}\).

Return result.

\[ \textbf{bf64\_NEGATE}(x) \]

\( x \) is a floating-point value represented in double-precision format.

Return \( x \) with its sign complemented.

\[ \textbf{bf64\_NEGATIVE\_ABSOLUTE}(x) \]

\( x \) is a floating-point value represented in double-precision format.

Return \( x \) with its sign set to 1.

\[ \textbf{bf64\_MAXIMUM}(x, y) \]

\( x \) is a binary floating-point value that is represented in double-precision format.
\( y \) is a binary floating-point value that is represented in double-precision format.

If \( x \) or \( y \) is an \( SNaN \), \( vx\_snan\_flag \) is set to 1.

If \( x \) is a \( QNaN \) and \( y \) is not a \( NaN \), return \( y \).
Otherwise, if \( x \) is a \( QNaN \), return \( x \).
Otherwise, if \( x \) is an \( SNaN \), return \( x \) represented as a \( QNaN \).
Otherwise, if \( y \) is a \( QNaN \), return \( x \).
Otherwise, if \( y \) is an \( SNaN \), return \( y \) represented as a \( QNaN \).
Otherwise, return the greater of \( x \) and \( y \), where \(+0\) is considered greater than \(-0\).

\[ \textbf{bf64\_MAXIMUM\_TYPE\_C}(x, y) \]

\( x \) is a binary floating-point value that is represented in double-precision format.
\( y \) is a binary floating-point value that is represented in double-precision format.

If \( x \) or \( y \) is an \( SNaN \), \( vx\_snan\_flag \) is set to 1.

If \( x \) or \( y \) is a \( NaN \), return \( y \).
Otherwise, if \( x \) is greater than \( y \), return \( x \).
Otherwise, return \( y \).
`bfp64_MAXIMUM_TYPE_J(x, y)`
- $x$ is a binary floating-point value that is represented in double-precision format.
- $y$ is a binary floating-point value that is represented in double-precision format.

If $x$ or $y$ is an SNaN, $vxsnan\_flag$ is set to 1.

If $x$ is a NaN, return $x$.
Otherwise, if $y$ is a NaN, return $y$.
Otherwise, if both $x$ and $y$ are Zero and either $x$ or $y$ is a +Zero, return +Zero.
Otherwise, if both $x$ and $y$ are Zero and both $x$ and $y$ are -Zero, return -Zero.
Otherwise, if $x$ is greater than $y$, return $x$.
Otherwise, return $y$.

`bfp64_MINIMUM(x, y)`
- $x$ is a binary floating-point value that is represented in double-precision format.
- $y$ is a binary floating-point value that is represented in double-precision format.

If $x$ or $y$ is an SNaN, $vxsnan\_flag$ is set to 1.

If $x$ is a QNaN and $y$ is not a NaN, return $y$.
Otherwise, if $x$ is a QNaN, return $x$.
Otherwise, if $x$ is an SNaN, return $x$ represented as a QNaN.
Otherwise, if $y$ is a QNaN, return $x$.
Otherwise, if $y$ is an SNaN, return $y$ represented as a QNaN.
Otherwise, return the lesser of $x$ and $y$, where -0 is considered less than +0.

`bfp64_MINIMUM_TYPE_C(x, y)`
- $x$ is a binary floating-point value that is represented in double-precision format.
- $y$ is a binary floating-point value that is represented in double-precision format.

If $x$ or $y$ is an SNaN, $vxsnan\_flag$ is set to 1.

If $x$ or $y$ is a NaN, return $y$.
Otherwise, if $x$ is less than $y$, return $x$.
Otherwise, return $y$.

`bfp64_MINIMUM_TYPE_J(x, y)`
- $x$ is a binary floating-point value that is represented in double-precision format.
- $y$ is a binary floating-point value that is represented in double-precision format.

If $x$ or $y$ is an SNaN, $vxsnan\_flag$ is set to 1.

If $x$ is a NaN, return $x$.
Otherwise, if $y$ is a NaN, return $y$.
Otherwise, if both $x$ and $y$ are Zero and either $x$ or $y$ is a -Zero, return -Zero.
Otherwise, if both $x$ and $y$ are Zero and both $x$ and $y$ are +Zero, return +Zero.
Otherwise, if $x$ is less than $y$, return $x$.
Otherwise, return $y$.

`bfp128_ABSOLUTE(x)`
- $x$ is a floating-point value represented in quad-precision format.

Return $x$ with its sign set to 0.
Chapter 7. Vector-Scalar Extension Facility

bfp128_CONVERT_FROM_BFP(x)

x is a quad-precision floating-point value that is represented in the binary floating-point working format.

If x is a QNaN,
   the contents of bit 0 of result are set to the value of x.sign,
   the contents of bits 1:15 of result are set to the value 0b111_1111_1111_1111, and
   the contents of bits 16:127 of result are set to the value of bits 1:112 of x.significand.

Otherwise, if x is a Zero,
   the contents of bit 0 of result are set to the value of x.sign, and
   the contents of bits 1:15 of result are set to the value 0b000_0000_0000_0000, and
   the contents of bits 16:127 of result are set to the value 0x0000_0000_0000_0000_0000_0000_0000.

Otherwise, if x is an Infinity,
   the contents of bit 0 of result are set to the value of x.sign, and
   the contents of bits 1:15 of result are set to the value 0b111_1111_1111_1111, and
   the contents of bits 16:127 of result are set to the value 0x0000_0000_0000_0000_0000_0000_0000.

Otherwise, do the following.
   If the exponent of x is less than -16382,
      the contents of bit 0 of result are set to the value of x.sign,
      the contents of bits 1:15 of result are set to the value 0b000_0000_0000_0000, and
      the contents of bits 16:127 of result are set to the value of bits 1:112 of the significand of x shifted right by N bits, where N is the value -16382 subtracted by the value of the exponent of x.
   Otherwise,
      the contents of bit 0 of result are set to the value of x.sign,
      the contents of bits 1:15 of result are set to the sum of the exponent of x and 16383, and
      the contents of bits 16:127 of result are set to the value of bits 1:112 of the significand of x.

Return result (i.e., x in quad-precision format).

bfp128_NEGATE(x)

x is a floating-point value represented in quad-precision format.

Return x with its sign complemented.

bfp128_NEGATIVE_ABSOLUTE(x)

x is a floating-point value represented in quad-precision format.

Return x with its sign set to 1.

si64_CONVERT_FROM_BFP(x)

x is an integer value represented in the binary floating-point working format.

Return the value x in signed doubleword integer format.

ui64_CONVERT_FROM_BFP(x)

x is an integer value represented in the binary floating-point working format.

Return the value x in 64-bit unsigned integer format.

bfp128_MAXIMUM_TYPE_C(x,y)

x is a binary floating-point value that is represented in quad-precision format.

y is a binary floating-point value that is represented in quad-precision format.

If x or y is an SNaN, vxsnan_flag is set to 1.

If x or y is a NaN, return y.

Otherwise, if x is greater than y, return x.

Otherwise, return y.
\textbf{bfp128\_MAXI\_NUM\_TYPE\_J}(x, y)

\textit{x} is a binary floating-point value that is represented in quad-precision format.
\textit{y} is a binary floating-point value that is represented in quad-precision format.

If \textit{x} or \textit{y} is an SNaN, \texttt{vxsnan\_flag} is set to 1.

If \textit{x} is a NaN, return \textit{x}.
Otherwise, if \textit{y} is a NaN, return \textit{y}.
Otherwise, if both \textit{x} and \textit{y} are Zero and either \textit{x} or \textit{y} is a +Zero, return +Zero.
Otherwise, if both \textit{x} and \textit{y} are Zero and both \textit{x} and \textit{y} are -Zero, return -Zero.
Otherwise, if \textit{x} is greater than \textit{y}, return \textit{x}.
Otherwise, return \textit{y}.

\textbf{bfp128\_MINI\_NUM\_TYPE\_C}(x, y)

\textit{x} is a binary floating-point value that is represented in quad-precision format.
\textit{y} is a binary floating-point value that is represented in quad-precision format.

If \textit{x} or \textit{y} is an SNaN, \texttt{vxsnan\_flag} is set to 1.

If \textit{x} or \textit{y} is a NaN, return \textit{y}.
Otherwise, if \textit{x} is less than \textit{y}, return \textit{x}.
Otherwise, return \textit{y}.

\textbf{bfp128\_MINI\_NUM\_TYPE\_J}(x, y)

\textit{x} is a binary floating-point value that is represented in quad-precision format.
\textit{y} is a binary floating-point value that is represented in quad-precision format.

If \textit{x} or \textit{y} is an SNaN, \texttt{vxsnan\_flag} is set to 1.

If \textit{x} is a NaN, return \textit{x}.
Otherwise, if \textit{y} is a NaN, return \textit{y}.
Otherwise, if both \textit{x} and \textit{y} are Zero and either \textit{x} or \textit{y} is a -Zero, return -Zero.
Otherwise, if both \textit{x} and \textit{y} are Zero and both \textit{x} and \textit{y} are +Zero, return +Zero.
Otherwise, if \textit{x} is less than \textit{y}, return \textit{x}.
Otherwise, return \textit{y}.

\textbf{EXTZ32}(x)

Result of extending the \textit{b}-bit value \textit{x} on the left with 32-\textit{b} zeros, forming a 32-bit value.

\begin{verbatim}
b ← LENGTH(x)
result.bit[0:31-b] ← 0
result.bit[32-b:31] ← x
\end{verbatim}

\textbf{EXTZ64}(x)

Result of extending the \textit{b}-bit value \textit{x} on the left with 64-\textit{b} zeros, forming a 64-bit value.

\begin{verbatim}
b ← LENGTH(x)
result.bit[0:63-b] ← 0
result.bit[64-b:63] ← x
\end{verbatim}

\textbf{EXTZ128}(x)

Result of extending the \textit{b}-bit value \textit{x} on the left with 128-\textit{b} zeros, forming a 128-bit value.

\begin{verbatim}
b ← LENGTH(x)
result.bit[0:127-b] ← 0
result.bit[128-b:127] ← x
\end{verbatim}
fprf_CLASS_BFP16(x)

\( x \) is a floating-point value represented in half-precision format.

Return the 5-bit code that specifies the sign and class of \( x \).

Return \(0b10001\) if \( x \) is a Quiet NaN.
Return \(0b01001\) if \( x \) is a negative infinity.
Return \(0b00101\) if \( x \) is a positive infinity.
Return \(0b01010\) if \( x \) is a negative zero.
Return \(0b00010\) if \( x \) is a positive zero.
Return \(0b11100\) if \( x \) is a negative denormal value as represented in half-precision format.
Return \(0b10100\) if \( x \) is a positive denormal value as represented in half-precision format.
Return \(0b01100\) if \( x \) is a negative normal value as represented in half-precision format.
Return \(0b00100\) if \( x \) is a positive normal value as represented in half-precision format.

fprf_CLASS_BFP32(x)

\( x \) is a floating-point value represented in single-precision format.

Return the 5-bit code that specifies the sign and class of \( x \).

Return \(0b10001\) if \( x \) is a Quiet NaN.
Return \(0b01001\) if \( x \) is a negative infinity.
Return \(0b00101\) if \( x \) is a positive infinity.
Return \(0b01010\) if \( x \) is a negative zero.
Return \(0b00010\) if \( x \) is a positive zero.
Return \(0b11100\) if \( x \) is a negative denormal value as represented in single-precision format.
Return \(0b10100\) if \( x \) is a positive denormal value as represented in single-precision format.
Return \(0b01100\) if \( x \) is a negative normal value as represented in single-precision format.
Return \(0b00100\) if \( x \) is a positive normal value as represented in single-precision format.

fprf_CLASS_BFP64(x)

\( x \) is a floating-point value represented in double-precision format.

Return the 5-bit code that specifies the sign and class of \( x \).

Return \(0b10001\) if \( x \) is a Quiet NaN.
Return \(0b01001\) if \( x \) is a negative infinity.
Return \(0b00101\) if \( x \) is a positive infinity.
Return \(0b01010\) if \( x \) is a negative zero.
Return \(0b00010\) if \( x \) is a positive zero.
Return \(0b11100\) if \( x \) is a negative denormal value as represented in double-precision format.
Return \(0b10100\) if \( x \) is a positive denormal value as represented in double-precision format.
Return \(0b01100\) if \( x \) is a negative normal value as represented in double-precision format.
Return \(0b00100\) if \( x \) is a positive normal value as represented in double-precision format.

fprf_CLASS_BFP128(x)

\( x \) is binary floating-point value that is represented in quad-precision format.

Return the 5-bit characterization of the sign and class of \( x \).

Return \(0b10001\) if \( x \) is a Quiet NaN.
Return \(0b01001\) if \( x \) is negative and an infinity.
Return \(0b00100\) if \( x \) is negative and a normal number.
Return \(0b11100\) if \( x \) is negative and a denormal number.
Return \(0b00101\) if \( x \) is negative and a zero.
Return \(0b00010\) if \( x \) is positive and a zero.
Return \(0b10100\) if \( x \) is positive and a denormal number.
Return \(0b00100\) if \( x \) is positive and a normal number.
IsInf(x)
Return 1 if x is an Infinity, otherwise return 0.

IsNaN(x)
Return 1 if x is either an SNaN or a QNaN, otherwise return 0.

IsNeg(x)
Return 1 if x is a negative, nonzero value, otherwise return 0.

IsSNaN(x)
Return 1 if x is an SNaN, otherwise return 0.

IsZero(x)
Return 1 if x is a Zero, otherwise return 0.

reset_xflags()
vxsnan_flag is set to 0.
vxiem_flag is set to 0.
vxiv_flag is set to 0.
vxisi_flag is set to 0.
vxzdz_flag is set to 0.
vxsqrt_flag is set to 0.
vxcvi_flag is set to 0.
xx_flag is set to 0.
ux_flag is set to 0.
xx_flag is set to 0.
ux_flag is set to 0.

SetFX(x)
x is one of the exception flags in the FPSCR.
If the contents of x is 0, FX and x are set to 1.

si128_CONVERT_FROM_BFP(x)
x is an integer value represented in the binary floating-point working format.
If x is a NaN,
vxcvi_flag is set to 1,
vxsnan_flag is set to 1 if x is an SNaN, and
return 0x8000_0000_0000_0000_0000_0000_0000_0000,
Otherwise, do the following.
Let rnd be the value x truncated to an integral value.
Let exponent be the unbiased exponent of rnd.
Let significand be the significand of rnd.
If rnd is greater than 2^{127}-1,
vxcvi_flag is set to 1, and
return 0x7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF.
Otherwise, if rnd is less than -2^{127},
vxcvi_flag is set to 1, and
return 0x8000_0000_0000_0000_0000_0000_0000_0000.
Otherwise,
xx_flag is set to 1 if rnd is inexact,
inc_flag is set to 0,
\textit{significand} is shifted right by the difference 127-\textit{exponent} with 0s shifted in, if \textit{rnd} is negative, \textit{significand} is negated, and return bits 0:127 of \textit{significand}.

\texttt{si32\_CHOP(x)}
\hspace{1em} \textit{x} is a signed integer value.

Return the rightmost 32 bits of \textit{x} in 32-bit signed integer format.

\texttt{si32\_CLAMP(x)}
\hspace{1em} \textit{x} is a signed integer value.

If \textit{x} is greater than \texttt{2^{31}-1}, \textit{result} is the value \texttt{2^{31}-1}, and \textit{SAT} is set to 1.

Otherwise, if \textit{x} is less than \texttt{-2^{31}}, \textit{result} is the value \texttt{-2^{31}}, and \textit{SAT} is set to 1.

Otherwise, \textit{result} is \textit{x}.

Return \textit{x} in 32-bit signed integer format.

\texttt{ui128\_CONVERT\_FROM\_BFP(x)}
\hspace{1em} \textit{x} is an integer value represented in the binary floating-point working format.

If \textit{x} is a NaN,
\hspace{2em} \texttt{vxcvi\_flag} is set to 1,
\hspace{2em} \texttt{vxsnan\_flag} is set to 1 if \textit{x} is an SNaN, and
\hspace{2em} return \texttt{0x0000_0000_0000_0000_0000_0000_0000_0000}.

Otherwise, do the following.
\hspace{2em} Let \texttt{rnd} be the value \textit{x} truncated to an integral value.
\hspace{2em} Let \texttt{exponent} be the unbiased exponent of \texttt{rnd}.
\hspace{2em} Let \textit{significand} be the significand of \texttt{rnd}.

If \texttt{rnd} is greater than \texttt{2^{128}-1},
\hspace{2em} \texttt{vxcvi\_flag} is set to 1, and
\hspace{2em} return \texttt{0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF}.

Otherwise, if \texttt{rnd} is less than 0,
\hspace{2em} \texttt{vxcvi\_flag} is set to 1, and
\hspace{2em} return \texttt{0x0000_0000_0000_0000_0000_0000_0000_0000}.

Otherwise,
\hspace{2em} \texttt{xx\_flag} is set to 1 if \texttt{rnd} is inexact,
\hspace{2em} \texttt{inc\_flag} is set to 0,
\hspace{2em} \textit{significand} is shifted right by the difference 127-\textit{exponent} with 0s shifted in, and
\hspace{2em} return bits 0:127 of \textit{significand}.
7.6.3 VSX Instruction Descriptions

**Load VSX Scalar Doubleword DS-form**

lxsd \( \text{VRT,DS(RA)} \)

<table>
<thead>
<tr>
<th>57</th>
<th>VRT</th>
<th>RA</th>
<th>DS</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

**Prefixed Load VSX Scalar Doubleword 8LS:D-form**

plxsd \( \text{VRT,D(RA),R} \)

Prefix:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>//</th>
<th>11</th>
<th>12</th>
<th>14</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>31</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th>42</th>
<th>VRT</th>
<th>RA</th>
<th>d1</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

If MSR.VSX=0 then VSX_Unavailable();

If "lxsd" then

\( EA \leftarrow (RA|0) + \text{EXT64}(\text{DS}||0b00) \)

If "plxsd" & R=0 then

\( EA \leftarrow (RA|0) + \text{EXT64}(d0||d1) \)

If "plxsd" & R=1 then

\( EA \leftarrow \text{CIA} + \text{EXT64}(d0||d1) \)

\( \text{VSR}[\text{VRT}+32].\text{dword}[0] \leftarrow \text{MEN}(EA,8) \)

\( \text{VSR}[\text{VRT}+32].\text{dword}[1] \leftarrow 0x0000_0000_0000_0000 \)

Let \( XT \) be the value \( \text{VRT} + 32 \).

For \( \text{lxsd} \), let the effective address \( (EA) \) be the sum of the contents of register \( RA \), or the value 0 if \( RA=0 \), and the value \( DS||0b00 \), sign-extended to 64 bits.

For \( \text{plxsd} \) with \( R=0 \), let the effective address \( (EA) \) be the sum of the contents of register \( RA \), or the value 0 if \( RA=0 \), and the value \( d0||d1 \), sign-extended to 64 bits.

For \( \text{plxsd} \) with \( R=1 \), let the effective address \( (EA) \) be the sum of the address of the instruction and the value \( d0||d1 \), sign-extended to 64 bits.

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address \( EA \) are placed into \( \text{load_data} \) in such an order that;

– the contents of the byte in storage at address \( EA \) are placed into byte 0 of \( \text{load_data} \),

– the contents of the byte in storage at address \( EA+1 \) are placed into byte 1 of \( \text{load_data} \), and so forth until

– the contents of the byte in storage at address \( EA+7 \) are placed into byte 7 of \( \text{load_data} \).

When Little-Endian byte ordering is employed, let \( \text{load_data} \) be the contents of the doubleword in storage at address \( EA \) such that;

– the contents of the byte in storage at address \( EA \) are placed into byte 7 of \( \text{load_data} \),

– the contents of the byte in storage at address \( EA+1 \) are placed into byte 6 of \( \text{load_data} \), and so forth until

– the contents of the byte in storage at address \( EA+7 \) are placed into byte 0 of \( \text{load_data} \).

\( \text{load_data} \) is placed into doubleword element 0 of \( \text{VSR}[\text{VRT}+32] \).

The contents of doubleword element 1 of \( \text{VSR}[\text{VRT}+32] \) are set to 0.

For \( \text{plxsd} \), if \( R \) is equal to 1 and \( RA \) is not equal to 0, the instruction form is invalid.

**Special Registers Altered:** None

**Extended Mnemonics:**

Extended mnemonics for **Prefixed Load VSX Scalar Doubleword**:

<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>plxsd ( \text{Vx}, \text{value}(\text{Ry}) )</td>
<td>plxsd ( \text{Vx}, \text{value}(\text{Ry}),0 )</td>
</tr>
<tr>
<td>plxsd ( \text{Vx}, \text{value} )</td>
<td>plxsd ( \text{Vx}, \text{value}(0),1 )</td>
</tr>
</tbody>
</table>

**VSR Data Layout for lxsd**

<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XT].dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>
Load VSX Scalar Doubleword Indexed X-form

lxsdx XT,RA,RB

\[
\begin{array}{cccccc}
0 & 31 & T & 11 & 16 & 21 & 588 \\
\end{array}
\]

if MSR.VSX=0 then VSX_Unavailable();

\[
\begin{align*}
EA & \leftarrow (RA=0) \ ? 0 : GPR[RA]) + GPR[RB] \\
VSR[32 \times TX + T].dword[0] & \leftarrow \text{MEM}(EA, 8) \\
VSR[32 \times TX + T].dword[1] & \leftarrow 0x0000_0000_0000_0000
\end{align*}
\]

Let XT be the value \(32 \times TX + T\).

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that:

- the contents of the byte in storage at address EA are placed into byte 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+7 are placed into byte 7 of load_data.

load_data is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered
None

VSR Data Layout for lxsdx

<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XT].dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>127</td>
<td></td>
<td>127</td>
</tr>
</tbody>
</table>

When Little-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that:

- the contents of the byte in storage at address EA are placed into byte 7 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 6 of load_data, and so forth until
- the contents of the byte in storage at address EA+7 are placed into byte 0 of load_data.
Load VSX Scalar as Integer Byte & Zero Indexed X-form

Ixsbzx  XT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>781</th>
</tr>
</thead>
</table>

if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

$EA \leftarrow (\text{RA}=0 ? 0 : \text{GPR}[RA]) + \text{GPR}[RB]$

$\text{VSR}[32 \times TX + T].dword[0] \leftarrow \text{EXTZ64}(\text{MEM}(EA,1))$
$\text{VSR}[32 \times TX + T].dword[1] \leftarrow 0x0000_0000_0000_0000$

Let $XT$ be the value $32 \times TX + T$.

Let the effective address ($EA$) be sum of the contents of $\text{GPR}[RA]$, or 0 if $RA$ is equal to 0, and the contents of $\text{GPR}[RB]$.

The unsigned integer in the byte in storage addressed by $EA$ is placed in doubleword element 0 of $\text{VSR}[XT]$. The contents of doubleword element 1 of $\text{VSR}[XT]$ are set to 0.

Special Registers Altered:
None

VSR Data Layout for Ixsbzx

<table>
<thead>
<tr>
<th>tgt</th>
<th>$\text{VSR}[XT].dword[0]$</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>

Load VSX Scalar as Integer Halfword & Zero Indexed X-form

Ixsihzx  XT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>813</th>
</tr>
</thead>
</table>

if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

$EA \leftarrow (\text{RA}=0 ? 0 : \text{GPR}[RA]) + \text{GPR}[RB]$

$\text{VSR}[32 \times TX + T].dword[0] \leftarrow \text{EXTZ64}(\text{MEM}(EA,2))$
$\text{VSR}[32 \times TX + T].dword[1] \leftarrow 0x0000_0000_0000_0000$

Let $XT$ be the value $32 \times TX + T$.

Let the effective address ($EA$) be sum of the contents of $\text{GPR}[RA]$, or 0 if $RA$ is equal to 0, and the contents of $\text{GPR}[RB]$.

The unsigned integer in the halfword in storage addressed by $EA$ is placed in doubleword element 0 of $\text{VSR}[XT]$. The contents of doubleword element 1 of $\text{VSR}[XT]$ are set to 0.

Special Registers Altered:
None

VSR Data Layout for Ixsihzx

<table>
<thead>
<tr>
<th>tgt</th>
<th>$\text{VSR}[XT].dword[0]$</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td>96</td>
</tr>
</tbody>
</table>
Load VSX Scalar as Integer Word Algebraic Indexed X-form

\texttt{lxsiwax XT,RA,RB}

\begin{verbatim}
if MSR.VSX=0 then VSX_Unavailable()
EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
VSR[32×TX+T].dword[0] ← EXTS64(MEM(EA,4))
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000

Let XT be the value \(32\times TX + T\).

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into \texttt{load_data} in such an order that;

– the contents of the byte in storage at address EA are placed into byte 0 of \texttt{load_data},
– the contents of the byte in storage at address EA+1 are placed into byte 1 of \texttt{load_data},
– the contents of the byte in storage at address EA+2 are placed into byte 2 of \texttt{load_data}, and
– the contents of the byte in storage at address EA+3 are placed into byte 3 of \texttt{load_data}.

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into \texttt{load_data} in such an order that;

– the contents of the byte in storage at address EA are placed into byte 3 of \texttt{load_data},
– the contents of the byte in storage at address EA+1 are placed into byte 2 of \texttt{load_data},
– the contents of the byte in storage at address EA+2 are placed into byte 1 of \texttt{load_data}, and
– the contents of the byte in storage at address EA+3 are placed into byte 0 of \texttt{load_data}.

\texttt{load_data} is sign-extended to a doubleword and placed in doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered
None

VSR Data Layout for \texttt{lxsiwax}

\begin{verbatim}
<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XT].dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{verbatim}
**Load VSX Scalar as Integer Word & Zero Indexed X-form**

```
lxsiwzx XT,RA,RB

if MSR.VSX=0 then VSX_Unavailable()  

EA ← (((RA=0) ? 0 : GPR[RA]) + GPR[RB])

VSR[32×TX+T].dword[0] ← ExtendZero(MEM(EA,4))
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte 2 of load_data, and
- the contents of the byte in storage at address EA+3 are placed into byte 3 of load_data.

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 3 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 2 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte 1 of load_data, and
- the contents of the byte in storage at address EA+3 are placed into byte 0 of load_data.

load_data is zero-extended and placed in doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

**Special Registers Altered**

None

**VSR Data Layout for lxsiwzx**

```
<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XT].dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
</table>
```
Load VSX Scalar Single-Precision DS-form

\( \text{lxssp} \quad \text{VRT}, \text{DS}(\text{RA}) \)

<table>
<thead>
<tr>
<th>57</th>
<th>VRT</th>
<th>RA</th>
<th>DS</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Prefixed Load VSX Scalar Single-Precision 8LS:D-form

\( \text{plxssp} \quad \text{VRT}, \text{D}(\text{RA}) \), \text{R} 

Prefix:

<table>
<thead>
<tr>
<th>43</th>
<th>VRT</th>
<th>RA</th>
<th>d1</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th>57</th>
<th>VRT</th>
<th>RA</th>
<th>DS</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

if "lxssp" then
   \( \text{EA} \leftarrow (\text{RA}|0) + \text{EXT64}(\text{DS}||0b00) \)
if "plxssp" & \text{R}=0 then
   \( \text{EA} \leftarrow (\text{RA}|0) + \text{EXT64}(d0||d1) \)
if "plxssp" & \text{R}=1 then
   \( \text{EA} \leftarrow \text{CIA} + \text{EXT64}(d0||d1) \)

\( \text{load\_data} \leftarrow \text{MEM}(\text{EA}, 4) \)
\( \text{result} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP32}(\text{MEM}(\text{EA}, 4)) \)
\( \text{VSR}[\text{VRT}+32].\text{dword}[0] \leftarrow \text{bfp64\_CONVERT\_FROM\_BFP}(\text{result}) \)
\( \text{VSR}[\text{VRT}+32].\text{dword}[1] \leftarrow 0x0000_0000_0000_0000 \)

Let \( \text{XT} \) be the value \( \text{VRT} \ + \ 32 \).

For \( \text{lxssp} \), let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value DS||0b00, sign-extended to 64 bits.

For \( \text{plxssp} \) with \( \text{R}=0 \), let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

For \( \text{plxssp} \) with \( \text{R}=1 \), let the effective address (EA) be the sum of the address of the instruction and the value d0||d1, sign-extended to 64 bits.

When Big-Endian byte ordering is employed, the contents of the word in storage at address \( \text{EA} \) are placed into \( \text{load\_data} \) in such an order that:

- the contents of the byte in storage at address \( \text{EA} \) are placed into byte 0 of \( \text{load\_data} \),
- the contents of the byte in storage at address \( \text{EA}+1 \) are placed into byte 1 of \( \text{load\_data} \),
- the contents of the byte in storage at address \( \text{EA}+2 \) are placed into byte 2 of \( \text{load\_data} \), and
- the contents of the byte in storage at address \( \text{EA}+3 \) are placed into byte 3 of \( \text{load\_data} \).

When Little-Endian byte ordering is employed, the contents of the word in storage at address \( \text{EA} \) are placed into \( \text{load\_data} \) in such an order that:

- the contents of the byte in storage at address \( \text{EA} \) are placed into byte 3 of \( \text{load\_data} \),
- the contents of the byte in storage at address \( \text{EA}+1 \) are placed into byte 2 of \( \text{load\_data} \),
- the contents of the byte in storage at address \( \text{EA}+2 \) are placed into byte 1 of \( \text{load\_data} \), and
- the contents of the byte in storage at address \( \text{EA}+3 \) are placed into byte 0 of \( \text{load\_data} \).

\( \text{load\_data} \), interpreted as a single-precision floating-point value, is placed into doubleword element 0 of \( \text{VSR}[\text{VRT}+32] \) in double-precision format.

The contents of doubleword element 1 of \( \text{VSR}[\text{VRT}+32] \) are set to 0.

For \( \text{plxssp} \), if \( \text{R} \) is equal to 1 and \( \text{RA} \) is not equal to 0, the instruction form is invalid.

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonics for \( \text{Prefixed Load VSX Scalar Single} \):

Extended mnemonic: Equivalent to:
\( \text{plxssp \ Vx, value(Ry)} \) \( \text{plxssp \ Vx, value(0), 0} \)
\( \text{plxssp \ Vx, value} \) \( \text{plxssp \ Vx, value(0), 1} \)
Load VSX Scalar Single-Precision Indexed X-form

\[
\text{lxssp} \quad XT, RA, RB
\]

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

load_data ← MEM(EA,4)
result ← bfp_CONVERT_FROM_BFP32(MEM(EA,4))
VSR[VT+32].dword[0] ← bfp64_CONVERT_FROM_BFP(result)
VSR[VT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32 × TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into \( \text{load} \_\text{data} \) in such an order that;

- the contents of the byte in storage at address EA are placed into byte 0 of \( \text{load} \_\text{data} \),
- the contents of the byte in storage at address EA+1 are placed into byte 1 of \( \text{load} \_\text{data} \),
- the contents of the byte in storage at address EA+2 are placed into byte 2 of \( \text{load} \_\text{data} \), and
- the contents of the byte in storage at address EA+3 are placed into byte 3 of \( \text{load} \_\text{data} \).

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into \( \text{load} \_\text{data} \) in such an order that;

- the contents of the byte in storage at address EA are placed into byte 3 of \( \text{load} \_\text{data} \),
- the contents of the byte in storage at address EA+1 are placed into byte 2 of \( \text{load} \_\text{data} \),
- the contents of the byte in storage at address EA+2 are placed into byte 1 of \( \text{load} \_\text{data} \), and
- the contents of the byte in storage at address EA+3 are placed into byte 0 of \( \text{load} \_\text{data} \).

\( \text{load} \_\text{data} \), interpreted as a single-precision floating-point value, is placed in doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered
None
**Load VSX Vector DQ-form**

`lxv XT,DQ(RA)`

<table>
<thead>
<tr>
<th>61</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>DQ</th>
</tr>
</thead>
</table>

**Prefixed Load VSX Vector 8LS:D-form**

`plxv XT,D(RA),R`  
Prefix:  
Suffix:  

| 25 | 11 | 16 | 31 |

if “lxv” & TX=0 & MSR.VSX=0 then VSX_Unavailable()  
if “lxv” & TX=1 & MSR.VEC=0 then Vector_Unavailable()  
if “plxv” & MSR.VSX=0 then VSX_Unavailable()  
if “lxv” then  

$$EA \leftarrow (RA|0) + EXTS64(DQ||0b0000)$$

if “plxv” & R=0 then  

$$EA \leftarrow (RA|0) + EXTS64(d0||d1)$$

if “plxv” & R=1 then  

$$EA \leftarrow (IA + EXTS64(d0||d1)$$

$$VSR[32\times TX+T] \leftarrow \text{MEM}(EA,16)$$  

Let $XT$ be the value $32\times TX + T$.  
For `lxv`, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value DQ||0b0000, sign-extended to 64 bits.  
For `plxv` with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.  
For `plxv` with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0||d1, sign-extended to 64 bits.

When Big-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into `$load_data$` in such an order that:

- the contents of the byte in storage at address EA are placed into byte element 0 of $load_data$,
- the contents of the byte in storage at address EA+1 are placed into byte element 1 of $load_data$, and so forth until
- the contents of the byte in storage at address EA+15 are placed into byte element 15 of $load_data$.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into `$load_data$` in such an order that:

- the contents of the byte in storage at address EA are placed into byte element 15 of $load_data$,
- the contents of the byte in storage at address EA+1 are placed into byte element 14 of $load_data$, and so forth until
- the contents of the byte in storage at address EA+15 are placed into byte element 0 of $load_data$.  

`load_data` is placed into $VSR[XT]$.  
For `plxv`, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

**Special Registers Altered**  
None

**Extended Mnemonics:**

Extended mnemonics for Prefixed Load VSX Vector:

<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plxv</code> Vx,value(Ry)`</td>
<td><code>plxv</code> Vx,value(Ry),0</td>
</tr>
<tr>
<td><code>plxv</code> Vx,value `</td>
<td><code>plxv</code> Vx,value(0),1</td>
</tr>
</tbody>
</table>

**VSR Data Layout for lxv**

<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XT]</th>
</tr>
</thead>
</table>
**Load VSX Vector Byte*16 Indexed X-form**

\texttt{lxvb16x XT,RA,RB}  

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>876</th>
<th>T</th>
</tr>
</thead>
</table>

if TX=0 & MSR.VSX=0 then VSX.Unavailable()  
if TX=1 & MSR.VEC=0 then Vector.Unavailable()  

\[ \text{EA} \leftarrow \left( (\text{RA}=0) \ ? \ 0 : \text{GPR}[\text{RA}] \right) + \text{GPR}[\text{RB}] \]

\[
\text{do } i = 0 \text{ to } 15 \\
\text{VSR}[32 \times TX + T].\text{byte}[i] \leftarrow \text{MEM}(\text{EA} + i, 1) \\
\text{end}
\]

Let XT be the value \(32 \times TX + T\).

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value from 0 to 15, do the following.

The contents of the byte in storage at address \(\text{EA} + i\) are placed into byte element \(i\) of VSR[XT],

**Special Registers Altered:**  
None

**Programming Note**

\texttt{lxvd2x}, \texttt{lxvw4x}, \texttt{lxvh8x}, \texttt{lxvb16x}, and \texttt{lxvx} exhibit identical behavior in Big-Endian mode.

---

**Example:** Loading data using **Load VSX Vector Byte*16 Indexed**

```plaintext
char X[] = { 0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0xE0, 0xE1, 0xE2, 0xE3, 0xE4, 0xE5, 0xE6, 0xE7 };
```

Big-endian storage image of X

```
addr[X]: F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
```

Little-endian storage image of X

```
addr[X]: F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
```

Loading a vector of 16 byte elements from Big-Endian storage in VSR[XT] using \texttt{lxvb16x}, retaining left-to-right element ordering.

```plaintext
lxvb16x xX,r0,rPX
```

VSR[W]:

```
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
```

Loading a vector of 16 byte elements from Little-Endian storage in VSR[XT] using \texttt{lxvb16x}, retaining left-to-right element ordering.

```plaintext
lxvb16x xX,r0,rPX
```

VSR[X]:

```
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
```

---

**VSR Data Layout for lxvb16x**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
<td>127</td>
</tr>
</tbody>
</table>
Load VSX Vector Doubleword*2 Indexed X-form

`lxvd2x XT,RA,RB`

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>844</th>
<th>11</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← MEM(EA, 8)
VSR[32×TX+T].dword[1] ← MEM(EA+8, 8)

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value i from 0 to 1, do the following.

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA+8×i are placed into `load_data` in such an order that;

- the contents of the byte in storage at address EA+8×i are placed into byte element 0 of `load_data`,
- the contents of the byte in storage at address EA+8×i+1 are placed into byte element 1 of `load_data`, and so forth until
- the contents of the byte in storage at address EA+8×i+7 are placed into byte element 7 of `load_data`.

`load_data` is placed into doubleword element i of VSR[XT].

Special Registers Altered

None

Programming Note

`lxvd2x, lxvv4x, lxvh8x, lxvb16x`, and `lxvx` exhibit identical behavior in Big-Endian mode.
Load VSX Vector Special Value Quadword X-form

let XT, UIM

<table>
<thead>
<tr>
<th>XT</th>
<th>UIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>360</td>
</tr>
<tr>
<td>11</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VSX = 0 then VSX_Unavailable()
if UIM = 0b00001 then VSR[32×TX + T] ← 0x3FFF_0000_0000_0000_0000_0000_0000_0000 /* QP +1.0 */
if UIM = 0b00010 then VSR[32×TX + T] ← 0x4000_0000_0000_0000_0000_0000_0000_0000 /* QP +2.0 */
if UIM = 0b00011 then VSR[32×TX + T] ← 0x4000_8000_0000_0000_0000_0000_0000_0000 /* QP +3.0 */
if UIM = 0b00100 then VSR[32×TX + T] ← 0x4001_0000_0000_0000_0000_0000_0000_0000 /* QP +4.0 */
if UIM = 0b00101 then VSR[32×TX + T] ← 0x4001_4000_0000_0000_0000_0000_0000_0000 /* QP +5.0 */
if UIM = 0b00110 then VSR[32×TX + T] ← 0x4001_8000_0000_0000_0000_0000_0000_0000 /* QP +6.0 */
if UIM = 0b00111 then VSR[32×TX + T] ← 0x4001_C000_0000_0000_0000_0000_0000_0000 /* QP +7.0 */
if UIM = 0b01000 then VSR[32×TX + T] ← 0x7FFF_0000_0000_0000_0000_0000_0000_0000 /* QP +Inf */
if UIM = 0b01001 then VSR[32×TX + T] ← 0x7FFF_8000_0000_0000_0000_0000_0000_0000 /* QP dQNaN */
if UIM = 0b10000 then VSR[32×TX + T] ← 0x8000_0000_0000_0000_0000_0000_0000_0000 /* QP -0.0 */
if UIM = 0b10001 then VSR[32×TX + T] ← 0xBFFF_0000_0000_0000_0000_0000_0000_0000 /* QP -1.0 */
if UIM = 0b10010 then VSR[32×TX + T] ← 0xC000_0000_0000_0000_0000_0000_0000_0000 /* QP -2.0 */
if UIM = 0b10011 then VSR[32×TX + T] ← 0xC000_8000_0000_0000_0000_0000_0000_0000 /* QP -3.0 */
if UIM = 0b10100 then VSR[32×TX + T] ← 0xC001_0000_0000_0000_0000_0000_0000_0000 /* QP -4.0 */
if UIM = 0b10101 then VSR[32×TX + T] ← 0xC001_4000_0000_0000_0000_0000_0000_0000 /* QP -5.0 */
if UIM = 0b10110 then VSR[32×TX + T] ← 0xC001_8000_0000_0000_0000_0000_0000_0000 /* QP -6.0 */
if UIM = 0b10111 then VSR[32×TX + T] ← 0xC001_C000_0000_0000_0000_0000_0000_0000 /* QP -7.0 */
if UIM = 0b11000 then VSR[32×TX + T] ← 0xFFFF_0000_0000_0000_0000_0000_0000_0000 /* QP -Inf */

let XT be the value 32×TX + T.

UIM specifies one of a set of common values that is placed into VSR[XT]. Unspecified values of UIM are reserved.

Special Registers Altered:
None

VSR Data Layout for lxvkq

tgt VSR[XT] 0 127
Load VSX Vector with Length X-form

```markdown
lxvl XT,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>269</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

- If \( TX=0 \) & \( MSR\text{.VSX}=0 \) then \( \text{VSX\_Unavailable()} \)
- If \( TX=1 \) & \( MSR\text{.VEC}=0 \) then \( \text{Vector\_Unavailable()} \)

\[
EA \leftarrow \begin{cases} \text{RA=0} & \Rightarrow 0 \\ GPR[RA] & \text{otherwise} \end{cases}
\]

\[
nb \leftarrow \text{EXTZ(GPR[RB].bit[0:7])}
\]

If \( nb>16 \) then \( nb \leftarrow 16 \)

\[
\text{load\_data} \leftarrow 0x0000_0000_0000_0000_0000_0000_0000_0000
\]

- If \( MSR\text{.LE} = 0 \) then // Big-Endian byte-ordering
  \[
  \text{load\_data.byte[0:nb-1]} \leftarrow \text{MEM}[EA,nb]
  \]
- Else // Little-Endian byte-ordering
  \[
  \text{load\_data.byte[16-nb:15]} \leftarrow \text{MEM}[EA,nb]
  \]

\[
\text{VSR}[32\times TX+T] \leftarrow \text{load\_data}
\]

Let \( XT \) be the value \( 32\times TX + T \).

Let the effective address (\( EA \)) be the contents of \( GPR[RA] \), or 0 if \( RA \) is equal to 0.

Let \( nb \) be the unsigned integer value in bits 0:7 of \( GPR[RB] \).

If \( nb \) is equal to 0, the storage access is not performed and the contents of \( VSR[XT] \) are set to 0.

Otherwise, when Big-Endian byte-ordering is employed, do the following.

- If \( nb \) less than 16, the contents of the \( nb \) bytes in storage starting at address \( EA \) are placed into the leftmost \( nb \) bytes of \( VSR[XT] \), and the contents of the rightmost 16-\( nb \) bytes of \( VSR[XT] \) are set to \( 0x00 \).

Otherwise, the contents of the quadword in storage at address \( EA \) are placed into \( VSR[XT] \).

Otherwise, when Little-Endian byte ordering is employed, do the following.

- If \( nb \) less than 16, the contents of the \( nb \) bytes in storage starting at address \( EA \) are placed into the rightmost \( nb \) bytes of \( VSR[XT] \) in byte-reversed order, and the contents of the leftmost 16-\( nb \) bytes of \( VSR[XT] \) are set to \( 0x00 \).

Otherwise, the contents of the quadword in storage at address \( EA \) are placed into \( VSR[XT] \) in byte-reversed order.

If the contents of bits 8:63 of \( GPR[RB] \) are not equal to 0, the results are boundedly undefined.

**Special Registers Altered:**

None

VSR Data Layout for lxvl

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
<td>127</td>
</tr>
</tbody>
</table>
Example: Loading less than 16-byte data into VSR using lxvl

```c
char      S[14] = "This is a TEST";
short     X[6]  = { 0xE0E1, 0xE2E3, 0xE4E5, 0xE6E7, 0xE8E9, 0xEAEB };
binary80  Z    = 0xF0F1F2F3F4F5F6F7F8F9
```

Loading less than 16-byte data from Big-Endian storage in VSR[XT] using `lxvl`.

Big-endian storage image of S, X, & Z

<table>
<thead>
<tr>
<th>addr(S)+0x0000:</th>
<th>F0 F1 F2 F3 F4 F5 F6 F7 00 00 00 00 00 00 00 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr(S)+0x0010:</td>
<td>E2 E3 E4 E5 E6 E7 E8 E9 EB EA FA FB F8 F9 F6 F5</td>
</tr>
<tr>
<td>addr(S)+0x0020:</td>
<td>F6 F7 F8 F9 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

Little-endian storage image of S, X, & Z

<table>
<thead>
<tr>
<th>addr(S)+0x0000:</th>
<th>T E S T E0 E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr(S)+0x0010:</td>
<td>E3 E2 E4 E5 E6 E7 E8 EB EA FA F8 F7 F6 F5 F4</td>
</tr>
<tr>
<td>addr(S)+0x0020:</td>
<td>F3 F2 F1 F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

# Assumptions

- GPR[NS] = 14 (length of S in # of bytes)
- GPR[NX] = 12 (length of X in # of bytes)
- GPR[NZ] = 10 (length of Z in # of bytes)
- GPR[PS] = address of S

add    rPX,rPS,rNS    # address of X
add    rPZ,rPX,rNX    # address of Z
sldi   rLS,rNS,56
sldi   rLX,rNX,56
sldi   rLZ,rNZ,56
lxvl   xS,rPS,56
lxvl   xX,rPX,56
lxvl   xZ,rPZ,56

VSR register image of S, X, & Z

<table>
<thead>
<tr>
<th>VSR[S]:</th>
<th>T E S T E0 E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[X]:</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7 E8 EA EB</td>
</tr>
<tr>
<td>VSR[Z]:</td>
<td>F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

Loading less than 16-byte data from Little-Endian storage in VSR[XT] using `lxvl`.

VSR register image of S, X, & Z

<table>
<thead>
<tr>
<th>VSR[S]:</th>
<th>00 00 T E S T E0 E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[X]:</td>
<td>00 00 00 00 EA EB E8 E9 E6 E7 E4 E5 E2 E3 E0 E1</td>
</tr>
<tr>
<td>VSR[Z]:</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

Little-endian storage image of S, X, & Z

<table>
<thead>
<tr>
<th>addr(S)+0x0000:</th>
<th>T E S T E1 E0</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr(S)+0x0010:</td>
<td>E3 E2 E4 E5 E6 E7 E8 EB EA FA F8 F7 F6 F5 F4</td>
</tr>
<tr>
<td>addr(S)+0x0020:</td>
<td>F3 F2 F1 F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

# Assumptions

- GPR[NS] = 14 (length of S in # of bytes)
- GPR[NX] = 12 (length of X in # of bytes)
- GPR[NZ] = 10 (length of Z in # of bytes)
- GPR[PS] = address of S

add    rPX,rPS,rNS    # address of X
add    rPZ,rPX,rNX    # address of Z
sldi   rLS,rNS,56
sldi   rLX,rNX,56
sldi   rLZ,rNZ,56
lxvl   xS,rPS,56
lxvl   xX,rPX,56
lxvl   xZ,rPZ,56

VSR register image of S, X, & Z

<table>
<thead>
<tr>
<th>VSR[S]:</th>
<th>00 00 T E S T E0 E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[X]:</td>
<td>00 00 00 00 EA EB E8 E9 E6 E7 E4 E5 E2 E3 E0 E1</td>
</tr>
<tr>
<td>VSR[Z]:</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
Load VSX Vector with Length Left-justified X-form

lxvll XT,RA,RB

if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← (RA=0) ? 0 : GPR[RA]
nb ← EXTZ(GPR[RB].bit[0:7])
if nb>16 then nb ← 16
if nb>0 then do i = 0 to nb-1
  VSR[32×TX+T].byte[i] ← MEM(EA+i,1)
end
if nb<16 then do i = nb to 15
  VSR[32×TX+T].byte[i] ← 0x00
end

Let XT be the value 32×TX + T.

Let the effective address (EA) be the contents of GPR[RA], or 0 if RA is equal to 0.

Let nb be the unsigned integer value in bits 0:7 of GPR[RB].

If nb is equal to 0, the storage access is not performed and the contents of VSR[XT] are set to 0.

Otherwise, do the following.
If nb less than 16, the contents of the nb bytes in storage starting at address EA are placed into the leftmost nb bytes of VSR[XT], and the contents of the rightmost 16-nb bytes of VSR[XT] are set to 0x00.

Otherwise, the contents of the quadword in storage at address EA are placed into VSR[XT].

Data is loaded from storage into VSR[XT] in Big-Endian byte ordering (i.e., the byte in storage at address EA is placed into byte element 0 of VSR[XT], the byte in storage at address EA+1 is placed in byte element 1 of VSR[XT], and so forth).

If the contents of bits 8:63 of GPR[RB] are not equal to 0, the results are boundedly undefined.

Special Registers Altered:
None

VSR Data Layout for lxvll

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
<td>127</td>
</tr>
</tbody>
</table>
Example: Loading less than 16-byte left-justified data

```plaintext
decimal X = +1234567890123456789;
decimal Y = -123456;
decimal Z = +1004966723510220;
```

Loading less than 16-byte data from storage in VSR[XT], left-justified, using `lxvll`.

<table>
<thead>
<tr>
<th>Initial state of VSRs X, Y, &amp; Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[X]: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>VSR[Y]: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>VSR[Z]: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Big-endian &amp; Little-Endian storage image of X, Y, &amp; Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X+0x0000: 12 34 56 78 90 12 34 56 78 9C 01 23 45 6D 01 00</td>
</tr>
<tr>
<td>X+0x0010: 49 66 72 35 10 22 0C 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

# Assumptions
#   GPR[NX] = 10 (length of X)
#   GPR[NY] = 4 (length of Y)
#   GPR[NZ] = 9 (length of Z)
#   GPR[PX] = address of X
#   GPR[PY] = address of Y = address of X + 10
#   GPR[PZ] = address of Z = address of X + 10 + 4

```plaintext
lxvll xX,rPX,rNX
lxvll xY,rPY,rNY
lxvll xZ,rPZ,rNZ
```

<table>
<thead>
<tr>
<th>Final state of VSRs X, Y, &amp; Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[X]: 01 34 67 78 90 12 34 56 78 9C 00 00 00 00 00 00</td>
</tr>
<tr>
<td>VSR[Y]: 01 23 45 6D 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>VSR[Z]: 01 00 49 66 72 35 10 22 0C 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
```
Load VSX Vector Paired DQ-form

**lxvp** XTp,DQ(RA)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>6</td>
<td>RA</td>
<td>16</td>
</tr>
<tr>
<td>DQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Prefixed Load VSX Vector Paired 8LS:D-form**

**plxvp** XTp,D(RA),R

**Prefix:**

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>d0</td>
</tr>
</tbody>
</table>

**Suffix:**

<table>
<thead>
<tr>
<th>58</th>
<th>6</th>
<th>16</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

EAbase ← \((RA=0) \ ? 0 : GPR[RA]\)

if “lxvp” then

EAdisp ← EXTS64(DQ || 0b0000)

if “plxvp” then

EAdisp ← EXTS64(d0 || d1)

if “lxvp” then EA ← EAbase + EAdisp
if “plxvp” & R=0 then EA ← EAbase + EAdisp
if “plxvp” & R=1 then EA ← CIA + EAdisp

load_data ← MEM(EA, 32)

VSR[32×TX+2×Tp] ← load_data.bit[0:127]
VSR[32×TX+2×Tp+1] ← load_data.bit[128:255]

Let XTp be the value 32×TX + 2×Tp (i.e., only even values of XTp can be encoded in the instruction).

Let EAbase be the contents of GPR[RA], or 0 if RA=0.

For **lxvp**, let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0, and the value DQ||0b0000, sign-extended to 64 bits.

For **plxvp**, if R=0, let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

For **plxvp**, if R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0||d1, sign-extended to 64 bits.

When Big-Endian byte ordering is employed, the contents of the octword in storage at address EA are placed into load_data in such an order that:

- the contents of the byte in storage at address EA are placed into byte 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+31 are placed into byte 31 of load_data.

When Little-Endian byte ordering is employed, the contents of the octword in storage at address EA are placed into load_data in such an order that:

- the contents of the byte in storage at address EA are placed into byte 31 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 30 of load_data, and so forth until
- the contents of the byte in storage at address EA+31 are placed into byte 0 of load_data.

Bits 0-127 of load_data are placed into VSR[XTp].

Bits 128-255 of load_data is placed into VSR[XTp+1].

For **plxvp**, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

**Special Registers Altered**

None

**Extended Mnemonics:**

Extended mnemonics for **Load VSX Vector Paired Prefixed**:

**Extended mnemonic:**

Equivalent to:

<table>
<thead>
<tr>
<th><strong>plxvp</strong> Rx,value(Ry)</th>
<th><strong>plxvp</strong> Rx,value(Ry),0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>plxvp</strong> Rx,value</td>
<td><strong>plxvp</strong> Rx,value(0),1</td>
</tr>
</tbody>
</table>

---

**Programming Note**

For best performance, RA should be word-aligned.
**Load VSX Vector Paired Indexed X-form**

```plaintext
lxvpx       XTp,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>6</th>
<th>6</th>
<th>333</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Tp</td>
<td>RA</td>
<td>RB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

if MSR.VSX=0 then VSX_Unavailable() 

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

load_data ← MEM(EA,32)

VSR[32×TX+2×Tp] ← load_data.bit[0:127]

VSR[32×TX+2×Tp+1] ← load_data.bit[128:255]

Let XTp be the value $32 \times TX + 2 \times Tp$ (i.e., only even values of $XTp$ can be encoded in the instruction).

Let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0, and the integer value in GPR[RB].

When Big-Endian byte ordering is employed, the contents of the octword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+31 are placed into byte 31 of load_data.

When Little-Endiа byte ordering is employed, the contents of the octword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 31 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 30 of load_data, and so forth until
- the contents of the byte in storage at address EA+31 are placed into byte 0 of load_data.

Bits 0-127 of load_data are placed into VSR[XTp].

Bits 128-255 of load_data is placed into VSR[XTp+1].

**Special Registers Altered:**

None

**Programming Note**

For best performance, EA should be word-aligned.

---

**VSR Data Layout for lxvpx**

```
<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XTp]</th>
<th>VSR[XTp+1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Load VSX Vector Rightmost Byte Indexed X-form

lxvrbx XT,RA,RB

if MSR.VSX=0 then VSX_Unavailable()

EA = (RA=0) ? 0 : GPR[RA] + GPR[RB]
VSR[32×TX+T] = EXTZ128(MEM(EA,1))

Let XT be the value of 32×TX + T.

Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].

Load the contents of the byte in storage at address EA into byte element 15 of VSR[XT]. The contents of byte elements 0-14 of VSR[XT] are set to 0.

Special Registers Altered
None

VSR Data Layout for lxvrbx

tgt 0x00_0000_0000_0000_0000_0000_0000_0000 .byte[15]
Load VSX Vector Rightmost Doubleword Indexed X-form

lxvrdx XT,RA,RB

if MSR.VSX=0 then VSX_Unavailable()

EA = (((RA=0) ? 0 : GPR[RA]) + GPR[RB])

VSR[32×TX+T] = EXT2128(MEM(EA,8))

Let XT be the value of 32×TX + T.

Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].

Load the contents of the doubleword in storage at address EA into doubleword element 0 of VSR[XT]. The contents of doubleword element 0 of VSR[XT] are set to 0.

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte 0 of load_data,
– the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data, and so forth until
– the contents of the byte in storage at address EA+7 are placed into byte 7 of load_data.

When Little-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte 7 of load_data,
– the contents of the byte in storage at address EA+1 are placed into byte 6 of load_data, and so forth until
– the contents of the byte in storage at address EA+7 are placed into byte 0 of load_data.

load_data is placed into doubleword element 1 of VSR[XT]. The contents of doubleword element 0 of VSR[XT] are set to 0.

Special Registers Altered

None

VSR Data Layout for lxvrdx

<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XT].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>
Load VSX Vector Rightmost Halfword Indexed X-form

\textbf{lxvrhx} \quad \text{XT,RA,RB}

\begin{verbatim}
if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
VSR[32\times TX+T] = EXTZ128(MEM(EA,2))
\end{verbatim}

Let \textit{XT} be the value of \(32 \times TX + T\).

Let \textit{EA} be the sum of \(GPR[RA]\), or 0 if \(RA=0\), and \(GPR[RB]\).

When Big-Endian byte ordering is employed, the contents of the halfword in storage at address \(EA\) are placed into \textit{load\_data} in such an order that;

\begin{itemize}
  \item the contents of the byte in storage at address \(EA\) are placed into byte 0 of \textit{load\_data}, and
  \item the contents of the byte in storage at address \(EA+1\) are placed into byte 1 of \textit{load\_data}.
\end{itemize}

When Little-Endian byte ordering is employed, the contents of the halfword in storage at address \(EA\) are placed into \textit{load\_data} in such an order that;

\begin{itemize}
  \item the contents of the byte in storage at address \(EA\) are placed into byte 1 of \textit{load\_data}, and
  \item the contents of the byte in storage at address \(EA+1\) are placed into byte 0 of \textit{load\_data}.
\end{itemize}

\textit{load\_data} is placed into halfword element 7 of \(VSR[XT]\).
The contents of halfword elements 0-6 of \(VSR[XT]\) are set to 0.

\textbf{Special Registers Altered}

\textit{None}

\textbf{VSR Data Layout for lxvrwx}

\begin{verbatim}
0x0000_0000_0000_0000_0000_0000_0000

\end{verbatim}

0 \quad 112 \quad 127
Load VSX Vector Rightmost Word Indexed X-form

\texttt{lxvrwx} \ XT,RA,RB

\begin{verbatim}

  31   T   RA   RB   77   77
   0   6   11   16   21   31

if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
VSR[32\times TX+T] = EXTZ128(MEM(EA,4))

Let XT be the value of 32\times TX + T.
Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load\_data in such an order that:

- the contents of the byte in storage at address EA are placed into byte 0 of load\_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load\_data,
- the contents of the byte in storage at address EA+2 are placed into byte 2 of load\_data,
- the contents of the byte in storage at address EA+3 are placed into byte 3 of load\_data, and
- the contents of the byte in storage at address EA+4 are placed into byte 4 of load\_data.

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load\_data in such an order that:

- the contents of the byte in storage at address EA are placed into byte 3 of load\_data,
- the contents of the byte in storage at address EA+1 are placed into byte 2 of load\_data,
- the contents of the byte in storage at address EA+2 are placed into byte 1 of load\_data, and
- the contents of the byte in storage at address EA+3 are placed into byte 0 of load\_data.

load\_data is placed into word element 3 of VSR[XT]. The contents of word elements 0-2 of VSR[XT] are set to 0.

Special Registers Altered
None

VSR Data Layout for lxvrwx

\begin{verbatim}

\textbf{tgt} \hspace{1cm} 0x0000_0000_0000_0000_0000_0000  \hspace{1cm} VSR[XT].word[3]

\end{verbatim}

\end{verbatim}

\end{verbatim}


Load VSX Vector Indexed X-form

\[ lxvx \quad XT,RA,RB \]

\[
\begin{array}{cccccccc}
31 & T & RA & RB & 4 & 12 & 21 & 16 \\
0 & 6 & 11 & 16 & 21 & 26 & 25 & 31 \\
\end{array}
\]

\[
\begin{align*}
\text{if } TX=0 & \text{ and } MSR.VSX=0 \text{ then } \text{VSX.Unavailable()} \\
\text{if } TX=1 & \text{ and } MSR.VEC=0 \text{ then } \text{Vector.Unavailable()} \\
EA & \leftarrow (\text{if } RA=0 \text{ then } 0 \text{ else } GPR[RA]) + GPR[RB] \\
\text{VSR}[32\times TX+T] & \leftarrow \text{MEM}(EA,16)
\end{align*}
\]

Let \( XT \) be the value \( 32\times TX + T \).

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+15 are placed into byte element 15 of load_data.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 15 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 14 of load_data, and so forth until
- the contents of the byte in storage at address EA+15 are placed into byte element 0 of load_data.

load_data is placed into VSR[XT].

Special Registers Altered:
None

VSR Data Layout for lxvx

\[
\begin{array}{cccc}
tgt & VSR[XT] & 127 \\
0 & & \\
\end{array}
\]
Example: Loading data using Load VSX Vector Indexed

```c
char W[16] = { 0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0x0E0, 0x0E1, 0x0E2, 0x0E3, 0x0E4, 0x0E5, 0x0E6, 0x0E7 };
short X[8] = { 0xF0F1, 0xF0F2, 0xF0F3, 0xF0F4, 0xF0F5, 0xF0F6, 0xF0F7, 0x0E0E1, 0x0E0E2, 0x0E0E3, 0x0E0E4, 0x0E0E5, 0x0E0E6, 0x0E0E7 };
float Y[4] = { 0xF0F1_F2F3_F4F5_F6F7, 0x0E0E1_E2E3_E4E5_E6E7 };
double Z[2] = { 0xF0F1_F2F3_F4F5_F6F7_F8F9_F0F1_F2F3_F4F5_F6F7_F8F9 };
```

Loading 16 bytes of data from Big-Endian storage in VSR[XT] using `lxvx`.

Big-endian storage image of W, X, Y, & Z

<table>
<thead>
<tr>
<th>addr(W+0x0000)</th>
<th>F0 F1 F2 F3 F4 F5 F6 F7</th>
<th>E0 E1 E2 E3 E4 E5 E6 E7</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr(W+0x0010)</td>
<td>F0 F1 F2 F3 F4 F5 F6 F7</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7</td>
</tr>
<tr>
<td>addr(W+0x0020)</td>
<td>F0 F1 F2 F3 F4 F5 F6 F7</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7</td>
</tr>
<tr>
<td>addr(W+0x0030)</td>
<td>F0 F1 F2 F3 F4 F5 F6 F7</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7</td>
</tr>
</tbody>
</table>

Final state of VSRs W, X, Y, & Z

VSR[W]:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
<td>F5</td>
<td>F6</td>
<td>F7</td>
<td>E0</td>
<td>E1</td>
<td>E2</td>
<td>E3</td>
<td>E4</td>
<td>E5</td>
<td>E6</td>
<td>E7</td>
</tr>
</tbody>
</table>

VSR[X]:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
<td>F5</td>
<td>F6</td>
<td>F7</td>
<td>E0</td>
<td>E1</td>
<td>E2</td>
<td>E3</td>
<td>E4</td>
<td>E5</td>
<td>E6</td>
<td>E7</td>
</tr>
</tbody>
</table>

VSR[Y]:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
<td>F5</td>
<td>F6</td>
<td>F7</td>
<td>E0</td>
<td>E1</td>
<td>E2</td>
<td>E3</td>
<td>E4</td>
<td>E5</td>
<td>E6</td>
<td>E7</td>
</tr>
</tbody>
</table>

VSR[Z]:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
<td>F5</td>
<td>F6</td>
<td>F7</td>
<td>E0</td>
<td>E1</td>
<td>E2</td>
<td>E3</td>
<td>E4</td>
<td>E5</td>
<td>E6</td>
<td>E7</td>
</tr>
</tbody>
</table>

Loading 16 bytes of data from Little-Endian storage in VSR[XT] using `lxvx`.

Little-endian storage image of W, X, Y, & Z

<table>
<thead>
<tr>
<th>addr(W+0x0000)</th>
<th>F0 F1 F2 F3 F4 F5 F6 F7</th>
<th>E0 E1 E2 E3 E4 E5 E6 E7</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr(W+0x0010)</td>
<td>F1 F0 F3 F2 F5 F4 F7 F6</td>
<td>E1 E0 E3 E2 E5 E4 E7 E6</td>
</tr>
<tr>
<td>addr(W+0x0020)</td>
<td>F2 F1 F0 F7 F6 F5 F4 F3</td>
<td>E2 E1 E0 E7 E6 E5 E4 E3</td>
</tr>
<tr>
<td>addr(W+0x0030)</td>
<td>F3 F2 F1 F0 F7 F6 F5 F4</td>
<td>E3 E2 E1 E0 E7 E6 E5 E4</td>
</tr>
</tbody>
</table>

Final state of VSRs W, X, Y, & Z

VSR[W]:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>E7</td>
<td>E6</td>
<td>E5</td>
<td>E4</td>
<td>E3</td>
<td>E2</td>
<td>E1</td>
<td>E0</td>
<td>F7</td>
<td>F6</td>
<td>F5</td>
<td>F4</td>
<td>F3</td>
<td>F2</td>
<td>F1</td>
<td>F0</td>
</tr>
</tbody>
</table>

VSR[X]:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>E6</td>
<td>E7</td>
<td>E4</td>
<td>E5</td>
<td>E2</td>
<td>E3</td>
<td>E0</td>
<td>F6</td>
<td>F7</td>
<td>F5</td>
<td>F4</td>
<td>F3</td>
<td>F0</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
</tr>
</tbody>
</table>

VSR[Y]:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>E4</td>
<td>E5</td>
<td>E6</td>
<td>E7</td>
<td>E0</td>
<td>E1</td>
<td>E2</td>
<td>F3</td>
<td>F4</td>
<td>F5</td>
<td>F6</td>
<td>F7</td>
<td>F0</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
</tr>
</tbody>
</table>

VSR[Z]:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0</td>
<td>E1</td>
<td>E2</td>
<td>E3</td>
<td>E4</td>
<td>E5</td>
<td>E6</td>
<td>E7</td>
<td>F0</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
<td>F5</td>
<td>F6</td>
<td>F7</td>
</tr>
</tbody>
</table>
Load VSX Vector Doubleword & Splat Indexed X-form

lxvdsx XT,RA,RB

If MSR.VSX=0 then VSX_Unavailable()

EA ← \begin{align*} & (\text{RA} = 0) \ ? \ 0 \ : \ \text{GPR}[\text{RA}] + \text{GPR}[\text{RB}] \\ & \end{align*}

load_data ← MEM(EA, 8)

\begin{align*} & \text{VSR}[32 \times TX + T].\text{dword}[0] \gets \text{load_data} \\ & \text{VSR}[32 \times TX + T].\text{dword}[1] \gets \text{load_data} \\ & \end{align*}

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of \text{GPR}[\text{RA}], or 0 if RA is equal to 0, and the contents of \text{GPR}[\text{RB}].

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that:

\begin{itemize}
  \item the contents of the byte in storage at address EA are placed into byte element 0 of load_data,
  \item the contents of the byte in storage at address EA+1 are placed into byte element 1 of load_data, and so forth until
  \item the contents of the byte in storage at address EA+7 are placed into byte element 7 of load_data.
\end{itemize}

When Little-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that:

\begin{itemize}
  \item the contents of the byte in storage at address EA are placed into byte element 7 of load_data,
  \item the contents of the byte in storage at address EA+1 are placed into byte element 6 of load_data, and so forth until
  \item the contents of the byte in storage at address EA+7 are placed into byte element 0 of load_data.
\end{itemize}

load_data is copied into each doubleword element of VSR[XT].

Special Registers Altered

None

VSR Data Layout for lxvdsx

<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XT].dword[0]</th>
<th>VSR[XT].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>
Load VSX Vector Halfword*8 Indexed X-form

\textit{lxvh8x} \ XT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>812</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

If TX=0 & MSR.VSX=0 then VSXUnavailable()
If TX=1 & MSR.VEC=0 then VectorUnavailable()

\[ EA \leftarrow ((RA=0) \ ? 0 : \text{GPR}[RA]) + \text{GPR}[RB] \]

\[
\text{do } i = 0 \text{ to } 7 \\
\text{VSR}[32\times TX + T].\text{hword}[i] \leftarrow \text{MEM}(EA+2\times i, 2) \\
\text{end}
\]

Let XT be the value $32 \times TX + T$.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value from 0 to 7, do the following.

When Big-Endian byte ordering is employed, the contents of the halfword in storage at address EA+2xi are placed into load_data in such an order that;

- the contents of the byte in storage at address EA+2xi are placed into byte element 0 of load_data,
- the contents of the byte in storage at address EA+2xi+1 are placed into byte element 1 of load_data.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into VSR[XT] in such an order that;

- the contents of the byte in storage at address EA+2xi are placed into byte element 1 of load_data,
- the contents of the byte in storage at address EA+2xi+1 are placed into byte element 0 of load_data.

load_data is placed into halfword element $i$ of VSR[XT].

Special Registers Altered:
None

VSR Data Layout for lxvh8x

\[ \begin{array}{cccccccc}
\text{tgt} & \text{VSR}[XT].\text{hword}[0] & \text{VSR}[XT].\text{hword}[1] & \text{VSR}[XT].\text{hword}[2] & \text{VSR}[XT].\text{hword}[3] & \text{VSR}[XT].\text{hword}[4] & \text{VSR}[XT].\text{hword}[5] & \text{VSR}[XT].\text{hword}[6] & \text{VSR}[XT].\text{hword}[7] \\
0 & 16 & 32 & 48 & 64 & 80 & 96 & 112 & 127 
\end{array} \]

---

Programming Note

\textit{lxvd2x}, \textit{lxvw4x}, \textit{lxvh8x}, \textit{lxvb16x}, and \textit{lxvx} exhibit identical behavior in Big-Endian mode.

Example: Loading data using Load VSX Vector Halfword*8 Indexed

\[
\text{short X[]} = \{ 0x0001, 0x1011, 0x2021, 0x3031, 0x4041, 0x5051, 0x6061, 0x7071 \}
\]

Big-endian storage image of X

\[
\begin{array}{cccccccc}
\text{addr}[X]: & 00 & 01 & 10 & 11 & 20 & 21 & 30 & 31 & 40 & 41 & 50 & 51 & 60 & 61 & 70 & 71 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F 
\end{array}
\]

Little-endian storage image of X

\[
\begin{array}{cccccccc}
\text{addr}[X]: & 00 & 01 & 11 & 10 & 21 & 20 & 31 & 30 & 41 & 40 & 51 & 50 & 61 & 60 & 71 & 70 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F 
\end{array}
\]

Loading a vector of 8 halfword elements from Big-Endian storage in VSR[XT] using \textit{lxvh8x}, retaining left-to-right element ordering.

\[
\begin{array}{cccccccc}
\text{VSR}[X]: & 00 & 01 & 10 & 11 & 20 & 21 & 30 & 31 & 40 & 41 & 50 & 51 & 60 & 61 & 70 & 71 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F 
\end{array}
\]

Loading a vector of 8 halfword elements from Little-Endian storage in VSR[XT] using \textit{lxvh8x}, retaining left-to-right element ordering.

\[
\begin{array}{cccccccc}
\text{VSR}[X]: & 00 & 01 & 10 & 11 & 20 & 21 & 30 & 31 & 40 & 41 & 50 & 51 & 60 & 61 & 70 & 71 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F 
\end{array}
\]
Load VSX Vector Word*4 Indexed X-form

**lxvw4x** XT,RA,RB

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>T</td>
<td>RA</td>
<td>RB</td>
<td>780</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].word[0] ← MEM(EA, 4)
VSR[32×TX+T].word[1] ← MEM(EA+4, 4)
VSR[32×TX+T].word[2] ← MEM(EA+8, 4)
VSR[32×TX+T].word[3] ← MEM(EA+12, 4)

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value i from 0 to 3, do the following.

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA+4×i are placed into load_data in such an order that;

- the contents of the byte in storage at address EA+4×i are placed into load_data, and
- the contents of the byte in storage at address EA+4×i+1 are placed into byte element 1 of load_data,
- the contents of the byte in storage at address EA+4×i+2 are placed into byte element 2 of load_data, and
- the contents of the byte in storage at address EA+4×i+3 are placed into byte element 3 of load_data.

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA+4×i are placed into word element i of VSR[XT] in such an order that;

- the contents of the byte in storage at address EA+4×i are placed into byte element 3 of load_data, and
- the contents of the byte in storage at address EA+4×i+1 are placed into byte element 2 of load_data,
- the contents of the byte in storage at address EA+4×i+2 are placed into byte element 1 of load_data, and
- the contents of the byte in storage at address EA+4×i+3 are placed into byte element 0 of load_data.

load_data is placed into word element i of VSR[XT].

Special Registers Altered
None

**Programming Note**

lxwd2x, lxvw4x, lxvh8x, lxvb16x, and lxvx exhibit identical behavior in Big-Endian mode.
Load VSX Vector Word & Splat Indexed X-form

lxvwsx XT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>364</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
load_data ← MEM(EA,4)
doi = 0 to 3
VSR[32×TX+T].word[i] ← load_data
end

Let XT be the value 32×TX + T.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 1 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte element 2 of load_data, and
- the contents of the byte in storage at address EA+3 are placed into byte element 3 of load_data.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 3 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 2 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte element 1 of load_data, and
- the contents of the byte in storage at address EA+3 are placed into byte element 0 of load_data.

load_data is copied into each word element of VSR[XT].

Special Registers Altered:
None

Example: Loading data using Load VSX Vector Word & Splat Indexed

int X = 0xF0F1_F2F3;

Big-endian storage image of X

addr[X]:
F0 F1 F2 F3
00 00 00 00 00 00 00 00 00 00 00 00
0123456789AB C D EF

Little-endian storage image of X

addr[X]:
F3 F2 F1 F0
00 00 00 00 00 00 00 00 00 00 00 00
0123456789AB C D EF

Loading scalar word data from Big-Endian storage in VSR[XT] using lxvwsx.

# Assumptions
# GPR(PX) = address of X

lxvwsx xX,r0,rPX

Final state of VSR X

VSR[X]:
F0 F1 F2 F3 F0 F1 F2 F3 F0 F1 F2 F3 F0 F1 F2 F3
0123456789AB C D EF

Loading scalar word data from Little-Endian storage in VSR[XT] using lxvwsx.

# Assumptions
# GPR(PX) = address of X

lxvwsx xX,r0,rPX

Final state of VSR X

VSR[X]:
F0 F1 F2 F3 F0 F1 F2 F3 F0 F1 F2 F3 F0 F1 F2 F3
0123456789AB C D EF
## VSR Data Layout for lxvwsx

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 3 26 49 6 1 2 7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Store VSX Scalar Doubleword DS-form

\[
\text{stxsd VRS,DS(RA)}
\]

<table>
<thead>
<tr>
<th></th>
<th>VRS</th>
<th>RA</th>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

Prefixed Store VSX Scalar Doubleword 8LS:D-form

\[
\text{pstxsd VRS,D(RA),R}
\]

Prefix:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th>46</th>
<th>VRS</th>
<th>RA</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable();

If "stxsd" then

\[
\text{EA} \leftarrow (\text{RA}|0) + \text{EXT64(DS||0b00)}
\]

If "pstxsd" & R=0 then

\[
\text{EA} \leftarrow (\text{RA}|0) + \text{EXT64(d0||d1)}
\]

If "pstxsd" & R=1 then

\[
\text{EA} \leftarrow (\text{CIA} + \text{EXT64(d0||d1)})
\]

\[
\text{MEM(EA,8)} \leftarrow \text{VSR[VRS+32].dword[0]}
\]

Let \(X_S\) be the value VRS + 32.

For **stxsd**, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value DS||0b00, sign-extended to 64 bits.

For **pstxsd** with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

For **pstxsd** with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0||d1, sign-extended to 64 bits.

Let \text{store_data} be the contents of doubleword element 0 of VSR[X_S].

When Big-Endian byte ordering is employed, \text{store_data} is placed in the doubleword in storage at address EA in such order that:

- byte 0 of \text{store_data} is placed into the byte in storage at address EA,
- byte 1 of \text{store_data} is placed into the byte in storage at address EA+1, and so forth until
- byte 7 of \text{store_data} is placed into the byte in storage at address EA+7.

When Little-Endian byte ordering is employed, \text{store_data} is placed in the doubleword in storage at address EA in such order that:

- the contents of byte 7 of doubleword element 0 of VSR[VRS+32] are placed into the byte in storage at address EA,
- the contents of byte 6 of doubleword element 0 of VSR[VRS+32] are placed into the byte in storage at address EA+1, and so forth until
- the contents of byte 0 of doubleword element 0 of VSR[VRS+32] are placed into the byte in storage at address EA+7.

For **pstxsd**, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for Prefixed Store VSX Scalar Doubleword:

<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pstxsd</strong> Vx, value(Ry)</td>
<td><strong>pstxsd</strong> Vx, value(Ry), 0</td>
</tr>
<tr>
<td><strong>pstxsd</strong> Vx, value</td>
<td><strong>pstxsd</strong> Vx, value(Ry), 1</td>
</tr>
</tbody>
</table>

VSR Data Layout for stxsd

<table>
<thead>
<tr>
<th>src</th>
<th>VSX[X_S].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>
Store VSX Scalar Doubleword Indexed X-form

```
stxsdx XS,RA,RB
```

if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,8) ← VSR[XS].dword[0]

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

Let store_data be the contents of doubleword element 0 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 7 of store_data is placed into the byte in storage at address EA+7.

When Little-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+7,
- byte 1 of store_data is placed into the byte in storage at address EA+6, and so forth until
- byte 7 of store_data is placed into the byte in storage at address EA.

**Special Registers Altered**

None

**VSR Data Layout for stxsdx**

```
src          VSX[XS].dword[0]      unused
0            64                   127
```
**Store VSX Scalar as Integer Byte Indexed X-form**

**stxsibx**  
**XS,RA,RB**

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>909</th>
<th>SX</th>
</tr>
</thead>
</table>

if SX=0 & MSR.VSX=0 then VSX_Unavailable()  
if SX=1 & MSR.VEC=0 then Vector_Unavailable()  

EA ← \( (RA=0) \ ? 0 : GPR[RA] \) + GPR[RB]  
MEM(EA,1) ← VSR[32×SX+S].byte[7]

Let XS be the value \( 32 \times SX + S \).

Let the effective address (EA) be sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of byte element 7 of VSR[XS] are placed into the byte in storage addressed by EA.

**Special Registers Altered:**  
None

---

**VSR Data Layout for stxsibx**

<table>
<thead>
<tr>
<th>src</th>
<th>unused</th>
<th>byte[7]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Store VSX Scalar as Integer Halfword Indexed X-form**

**stxsihx**  
**XS,RA,RB**

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>941</th>
<th>SX</th>
</tr>
</thead>
</table>

if SX=0 & MSR.VSX=0 then VSX_Unavailable()  
if SX=1 & MSR.VEC=0 then Vector_Unavailable()  

EA ← \( (RA=0) \ ? 0 : GPR[RA] \) + GPR[RB]  
MEM(EA,2) ← VSR[32×SX+S].hword[3]

Let XS be the value \( 32 \times SX + S \).

Let the effective address (EA) be sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of halfword element 3 of VSR[XS] are placed into the halfword in storage addressed by EA.

**Special Registers Altered:**  
None

---

**VSR Data Layout for stxsihx**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Store VSX Scalar as Integer Word Indexed X-form

\[
\text{stxsiwx} \quad \text{XS, RA, RB}
\]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>RA</th>
<th>RB</th>
<th>140</th>
<th>S</th>
</tr>
</thead>
</table>

If MSR.VSX=0 then VSX_Unavailable()

\[
\text{EA} \leftarrow \begin{cases} 
0 & \text{if } RA=0 \\
\text{GPR}[RA] + \text{GPR}[RB] & \text{otherwise}
\end{cases}
\]

\[
\text{MEM}(\text{EA}, 4) \leftarrow \text{VSR}[32 \times \text{SX} + S].\text{word}[1]
\]

Let XS be the value \(32 \times SX + S\).

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

Let store_data be the contents of word element 1 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that:

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1,
- byte 2 of store_data is placed into the byte in storage at address EA+2, and
- byte 3 of store_data is placed into the byte in storage at address EA+3.

Special Registers Altered
None

VSR Data Layout for stxsiwx

<table>
<thead>
<tr>
<th>src</th>
<th>unused</th>
<th>VSR[XS].word[1]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>
**Store VSX Scalar Single-Precision DS-form**

\[ \text{stxssp} \quad \text{VRS,DS(RA)} \]

<table>
<thead>
<tr>
<th>61</th>
<th>VRS</th>
<th>11</th>
<th>RA</th>
<th>16</th>
<th>DS</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>16</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Prefixed Store VSX Scalar Single-Precision 8LS:D-form**

\[ \text{pstxssp} \quad \text{VRS,D(RA),R} \]

**Prefix:**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>//</th>
<th>//</th>
<th>d0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

**Suffix:**

<table>
<thead>
<tr>
<th>47</th>
<th>VRS</th>
<th>11</th>
<th>RA</th>
<th>15</th>
<th>d1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>15</td>
<td>15</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- If “stxssp” then
  - \( \text{EA} \leftarrow (\text{RA} | 0) + \text{EXT64(DS||0b00)} \)
- If “pstxssp” & R=0 then
  - \( \text{EA} \leftarrow (\text{RA} | 0) + \text{EXT64(d0|d1)} \)
- If “pstxssp” & R=1 then
  - \( \text{EA} \leftarrow \text{CIA} + \text{EXT64(d0|d1)} \)

\[ \text{MEM(\text{EA},4) \leftarrow bfp32\_CONVERT\_FROM\_BFP64(VSR[\text{XS}].\text{dword}[0])} \]

Let \( \text{XS} \) be the value VRS + 32.

For **stxssp**, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value DS||0b00, sign-extended to 64 bits.

For **pstxssp** with R=0, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

For **pstxssp** with R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0||d1, sign-extended to 64 bits.

Let \( \text{store\_data} \) be the double-precision floating-point value in doubleword element 0 of VSR[XS] converted to single-precision format.

When Big-Endian byte ordering is employed, \( \text{store\_data} \) is placed in the word in storage at address EA in such order that:

- byte 0 of \( \text{store\_data} \) is placed into the byte in storage at address EA,
- byte 1 of \( \text{store\_data} \) is placed into the byte in storage at address EA+1,
- byte 2 of \( \text{store\_data} \) is placed into the byte in storage at address EA+2, and
- byte 3 of \( \text{store\_data} \) is placed into the byte in storage at address EA+3.

When Little-Endian byte ordering is employed, \( \text{store\_data} \) is placed in the word in storage at address EA in such order that:

- byte 0 of \( \text{store\_data} \) is placed into the byte in storage at address EA+3,
- byte 1 of \( \text{store\_data} \) is placed into the byte in storage at address EA+2,
- byte 2 of \( \text{store\_data} \) is placed into the byte in storage at address EA+1, and
- byte 3 of \( \text{store\_data} \) is placed into the byte in storage at address EA.

For **pstxssp**, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Extended mnemonics for Prefixed Store VSX Scalar Single-Precision:

<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{psxssp} ) Vx, value(R_y)</td>
<td>( \text{psxssp} ) Vx, value(R_y), 0</td>
</tr>
<tr>
<td>( \text{psxssp} ) Vx, value</td>
<td>( \text{psxssp} ) Vx, value(0), 1</td>
</tr>
</tbody>
</table>

**VSR Data Layout for stxssp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XS].dword[0]</th>
<th>unused</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Store VSX Scalar Single-Precision Indexed X-form

\[
\text{Store VSX Scalar Single-Precision Indexed X-form} \quad \text{stxsspx} \quad \text{XS,RA,RB}
\]

When Little-Endian byte ordering is employed, \( \text{store_data} \) is placed in the word in storage at address \( \text{EA} \) in such order that;

- byte 0 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+3 \),
- byte 1 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+2 \),
- byte 2 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+1 \), and
- byte 3 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA} \).

**Special Registers Altered**

None

<table>
<thead>
<tr>
<th>VSR Data Layout for stxsspx</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

When Big-Endian byte ordering is employed, \( \text{store_data} \) is placed in the word in storage at address \( \text{EA} \) in such order that;

- byte 0 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA} \),
- byte 1 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+1 \),
- byte 2 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+2 \), and
- byte 3 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+3 \).

\[
\begin{array}{c|c|c}
0 & 64 & 127 \\
\end{array}
\]
Store VSX Vector DQ-form

\[ \text{sbvx} \quad \text{XS}, \text{DQ}(\text{RA}) \]

<table>
<thead>
<tr>
<th>61</th>
<th>S</th>
<th>RA</th>
<th>DQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

Prefix:  stxv

Suffix:  if \( \text{stxv} \) & SX=0 & MSR.VSX=0 then VSX_Unavailable()
if \( \text{stxv} \) & SX=1 & MSR.VEC=0 then Vector_Unavailable()
if \( \text{stxv} \) then
\[ \text{EA} \leftarrow (\text{RA}|0) + \text{EXTS64(DQ}||0b0000) \]

MEM(\text{EA}, 16) \leftarrow \text{VSR}[32 \times \text{SX} + \text{S}]

Let \( \text{store\_data} \) be the contents of \( \text{VSR}[\text{XS}] \).

When Big-Endian byte ordering is employed, \( \text{store\_data} \) is placed into the quadword in storage at address \( \text{EA} \) in such an order that:

- byte 0 of \( \text{store\_data} \) is placed into the byte in storage at address \( \text{EA} \),
- byte 1 of \( \text{store\_data} \) is placed into the byte in storage at address \( \text{EA}+1 \), and so forth until
- byte 15 of \( \text{store\_data} \) is placed into the byte in storage at address \( \text{EA}+15 \).

When Little-Endian byte ordering is employed, \( \text{store\_data} \) is placed into the quadword in storage at address \( \text{EA} \) in such an order that:

- byte 15 of \( \text{store\_data} \) is placed into the byte in storage at address \( \text{EA} \),
- byte 14 of \( \text{store\_data} \) is placed into the byte in storage at address \( \text{EA}+1 \), and so forth until
- byte 0 of \( \text{store\_data} \) is placed into the byte in storage at address \( \text{EA}+15 \).

For \( \text{pstxv} \), if \( R \) is equal to 1 and RA is not equal to 0, the instruction form is invalid.

Special Registers Altered

None

Extended Mnemonics:

Extended mnemonics for Prefixed Store VSX Vector:

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{pstxv} \text{Vx}, \text{value}(\text{Ry}) )</td>
<td>( \text{pstxv} \text{Vx}, \text{value}(\text{Ry}), 0 )</td>
</tr>
<tr>
<td>( \text{pstxv} \text{Vx}, \text{value} )</td>
<td>( \text{pstxv} \text{Vx}, \text{value}(0), 1 )</td>
</tr>
</tbody>
</table>

VSR Data Layout for stxv

\[ \text{src} \quad \text{VSR}[\text{XS}] \]
**Store VSX Vector Byte*16 Indexed X-form**

\[
stxvb16x \quad XS, RA, RB
\]

if \( Sx = 0 \) & MSR.VSX=0 then VSX_Unavailable();
if \( Sx = 1 \) & MSR.VEC=0 then Vector_Unavailable();

\[
EA \leftarrow \begin{cases} 
0 & \text{if } RA = 0 \vspace{1pt} \text{or} \vspace{1pt} 0 : \text{GPR}[RA] + \text{GPR}[RB] \\
GPR[RB] & \text{otherwise}
\end{cases}
\]

\begin{align*}
do i = 0 \text{ to } 15 \\
\text{MEM}(EA+i, 1) & \leftarrow \text{VSR}[32 \times SX + S].\text{byte}[i]
\end{align*}

end

Let \( XS \) be the value \( 32 \times SX + S \).

Let the effective address (\( EA \)) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value from 0 to 15, do the following.

The contents of byte element \( i \) of VSR[XS] are placed into the byte in storage at address \( EA+i \).

**Special Registers Altered:** None

---

**Programming Note**

\( \text{stxvd2x, stxvw4x, stxvh8x, stxvb16x, and stxvx} \) exhibit identical behavior in Big-Endian mode.

---

**Example:** Storing data using Store VSX Vector Byte*16 Indexed

char \( X[16] \):

\[
\begin{array}{cccccccccccccccc}
& F0 & F1 & F2 & F3 & F4 & F5 & F6 & E0 & E1 & E2 & E3 & E4 & E5 & E6 & E7 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F
\end{array}
\]

Storing a vector of 16 byte elements from VSR[XS] into Big-Endian storage using \( \text{sxvb16x} \), retaining left-to-right element ordering.

\# Assumptions
\# GPR[PX] = address of \( X \)

\[
\text{sxvb16x} \quad xX, r0, rPX
\]

Big-endian storage image of \( X \)

\[
\begin{array}{cccccccccccccccc}
& F0 & F1 & F2 & F3 & F4 & F5 & F6 & E0 & E1 & E2 & E3 & E4 & E5 & E6 & E7 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F
\end{array}
\]

---

**VSR Data Layout for stxvb16x**

\[
src \begin{array}{cccccccccccc}
0 & 8 & 16 & 24 & 32 & 40 & 48 & 56 & 64 & 72 & 80 & 88 & 96 & 104 & 112 & 120 & 127
\end{array}
\]
Store VSX Vector Doubleword*2 Indexed X-form

stxvd2x XS,RA,RB

if MSR.VSX=0 then VSX_Unavailable();

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
MEM(EA,8) ← VSR[32×SX+S].dword[0]
MEM(EA+8,8) ← VSR[32×SX+S].dword[1]

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value i from 0 to 1, do the following.

Let store_data be the contents of doubleword element i of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA+8×i in such order that;

– byte 0 of store_data is placed into the byte in storage at address EA+8×i,
– byte 1 of store_data is placed into the byte in storage at address EA+8×i+1, and so forth until
– byte 7 of store_data is placed into the byte in storage at address EA+8×i+7.

When Little-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA+8×i in such order that;

– byte 0 of store_data is placed into the byte in storage at address EA+8×i+7,
– byte 1 of store_data is placed into the byte in storage at address EA+8×i+6, and so forth until
– byte 7 of store_data is placed into the byte in storage at address EA+8×i.

Special Registers Altered
None

Programming Note
stxvd2x, stxvw4x, stxvh8x, stxvb16x, and stxvx exhibit identical behavior in Big-Endian mode.

VSR Data Layout for stxvd2x

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XS].dword[0]</th>
<th>VSR[XS].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Store VSX Vector Halfword*8 Indexed X-form

stxvh8x XS,RA,RB

<table>
<thead>
<tr>
<th>S</th>
<th>RA</th>
<th>RB</th>
<th>940</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>8</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← (RA=0 ? 0 : GPR[RA]) + GPR[RB]

do i = 0 to 7
  MEM(EA+2×i,2) ← VSR[32×SX+S].hword[i]
end

Let XS be the value 32×SX + S.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value from 0 to 7, do the following.
  The contents of byte element i of VSR[XS] are placed into the byte in storage at address EA+i.

For each integer value from 0 to 7, do the following.
  When Big-Endian byte ordering is employed, the contents of halfword element i of VSR[XS] are placed into the halfword in storage at address EA+2×i in such an order that;
    – the contents of byte sub-element 0 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i, and
    – the contents of byte sub-element 1 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i + 1.

When Little-Endian byte ordering is employed, the contents of halfword element i of VSR[XS] are placed into the halfword in storage at address EA+2×i in such an order that;
    – the contents of byte sub-element 1 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i, and
    – the contents of byte sub-element 0 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i + 1.

Special Registers Altered:
None

Example: Storing data using Store VSX Vector Halfword*8 Indexed

stxvh8x xX,r0,rPX

Big-endian storage image of X

addr(X): 00 01 10 11 20 21 30 31 40 41 50 51 60 61 70 71
0 1 2 3 4 5 6 7 8 9 A B C D E F

Storing a vector of 8 halfword elements from VSR[X] into Big-Endian storage using stxvh8x, retaining left-to-right element ordering.

# Assumptions
# GPR[PX] = address of X
stxvh8x xX,r0,rPX

Little-endian storage image of X

addr(X): 01 00 11 10 21 20 31 30 41 40 51 50 61 60 71 70
0 1 2 3 4 5 6 7 8 9 A B C D E F

Storing a vector of 8 halfword elements from VSR[X] into Little-Endian storage using stxvh8x, retaining left-to-right element ordering.

# Assumptions
# GPR[PX] = address of X
stxvh8x xX,r0,rPX

Programming Note

stxvd2x, stxvw4x, stxvh8x, stxvb16x and stxvx exhibit identical behavior in Big-Endian mode.
Store VSX Vector with Length X-form

\[ \text{stxvl} \quad \text{XS}, \text{RA}, \text{RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>S</th>
<th>RA</th>
<th>RB</th>
<th>397</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if \( SX=0 \) & MSR.VSX=0 then VSX_Unavailable()
if \( SX=1 \) & MSR.VEC=0 then Vector_Unavailable()

\[ EA \leftarrow (RA=0) ? 0 : \text{GPR}[RA] \]
\[ nb \leftarrow \text{EXTZ} (\text{GPR}[RB].\text{bit}[0:7]) \]
if \( nb \geq 16 \) then \( nb \leftarrow 16 \)

if MSR.LE = 0 then       // Big-Endian byte-ordering
\[ \text{store_data} \leftarrow \text{VSR}[32 \times SX+S].\text{byte}[0:nb-1] \]
else                     // Little-Endian byte ordering
\[ \text{store_data} \leftarrow \text{VSR}[32 \times SX+S].\text{byte}[16-nb:15] \]

\[ \text{MEM}(EA,nb) \leftarrow \text{store_data} \]

Let \( XS \) be the value \( 32 \times SX + S \).

Let the effective address (\( EA \)) be the contents of \( \text{GPR}[RA] \), or 0 if \( RA \) is equal to 0.

Let \( nb \) be the unsigned integer value in bits 0:7 of \( \text{GPR}[RB] \).

If \( nb \) is equal to 0, the storage access is not performed.

VSR Data Layout for stxvl

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
<td>127</td>
</tr>
</tbody>
</table>

Otherwise, when Big-Endian byte-ordering is employed, do the following.
If \( nb \) less than 16, the contents of the leftmost \( nb \) bytes of \( \text{VSR}[XS] \) are placed in storage starting at address \( EA \).

Otherwise, the contents of \( \text{VSR}[XS] \) are placed into the quadword in storage at address \( EA \).

Otherwise, when Little-Endian byte ordering is employed, do the following.
If \( nb \) less than 16, the contents of the rightmost \( nb \) bytes of \( \text{VSR}[XS] \) are placed in storage starting at address \( EA \) in byte-reversed order.

Otherwise, the contents of \( \text{VSR}[XS] \) are placed into the quadword in storage at address \( EA \) in byte-reversed order.

If the contents of bits 8:63 of \( \text{GPR}[RB] \) are not equal to 0, the results are boundedly undefined.

Special Registers Altered:
None
Example: Storing less than 16-byte data from VSR using stxvl

```c
char S[14] = "This is a TEST"
short X[6] = { 0xE0E1, 0xE2E3, 0xE4E5, 0xE6E7, 0xE8E9, 0xEAEB }
binary80 Z = 0xF0F1F2F3F4F5F6F7F8F9
```

Storing less than 16-byte data in VSR[XS] into Big-Endian storage using stxvl.

```
# Assumptions
#   GPR[NS] = 14 (length of S in # of bytes)
#   GPR[NX] = 12 (length of X in # of bytes)
#   GPR[NZ] = 10 (length of Z in # of bytes)
#   GPR[PS] = address of S
```

VSR register image of S, X, & Z

```
VSR[S]:
00 00 "TS" "TS" TS "TS" TE "TS" TO 00 00
VSR[X]:
00 00 00 00 00 00 00 00 00 00 00 00
VSR[Z]:
00 00 00 00 00 00 00 00 00 00 00 00
```

Final state of Big-Endian storage image of S, X, & Z

```
addr(S)+0x0000:
01 02 03 04 05 06 07 08 09 A B C D E F
addr(S)+0x0010:
F6 F7 F8 F9 00 00 00 00 00 00 00 00 00 00
addr(S)+0x0020:
01 02 03 04 05 06 07 08 09 A B C D E F
```

Storing less than 16-byte data in VSR[XS] into Little-Endian storage using stxvl.

```
# Assumptions
#   GPR[NS] = 14 (length of S in # of bytes)
#   GPR[NX] = 12 (length of X in # of bytes)
#   GPR[NZ] = 10 (length of Z in # of bytes)
#   GPR[PS] = address of S
```

VSR register image of S, X, & Z

```
VSR[S]:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
VSR[X]:
00 00 00 00 EA E9 E6 E5 E4 E3 E2 E1 00 00 00 00 00 00 00
VSR[Z]:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Final state of Little-Endian storage image of S, X, & Z

```
addr(S)+0x0000:
E1 E0 E9 E8 E7 E6 E5 E4 E3 E2 EA F8 F7 F6 F5 F4 F3 F2 F1 F9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
addr(S)+0x0010:
E1 E0 E9 E8 E7 E6 E5 E4 E3 E2 EA F8 F7 F6 F5 F4 F3 F2 F1 F9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
addr(S)+0x0020:
E1 E0 E9 E8 E7 E6 E5 E4 E3 E2 EA F8 F7 F6 F5 F4 F3 F2 F1 F9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
add rPX,rPS,rNS  # address of X
add rPZ,rPX,rNX  # address of Z
sldi rLS,rNS,56  # address of X
sldi rLX,rNX,56  # address of Z
stxvl xS,rPS,rLS
stxvl xX,rPX,rLX
stxvl xZ,rPZ,rLZ
```

Final state of Big-Endian storage image of S, X, & Z
Store VSX Vector with Length Left-justified X-form

\[
\text{stxvll} \quad \text{XS,RA,RB}
\]

- If \( SX=0 \) & MSR.VSX=0 then VSX_Unavailable()
- If \( SX=1 \) & MSR.VEC=0 then Vector_Unavailable()

\[
\text{EA} \leftarrow (\text{RA}=0) \oplus 0 : \text{GPR}[\text{RA}]
\]

\[
nb \leftarrow \text{EXTZ}([\text{GPR}[\text{RB}].\text{bit}[0:7]])
\]

\[
\text{if } nb>16 \text{ then } nb \leftarrow 16
\]

\[
\text{if } nb>0 \text{ then do i = 0 to } nb-1
\]

\[
\text{MEM}(\text{EA}+i,1) \leftarrow \text{VSR}[32\timesSX+S].\text{byte}[i]
\]

Let \( XS \) be the value \( 32\times SX + S \).

Let the effective address (EA) be the contents of GPR[RA], or 0 if RA is equal to 0.

Let \( nb \) be the unsigned integer value in bits 0:7 of GPR[RB].

If \( nb \) is equal to 0, the storage access is not performed.

Otherwise, do the following.

- If \( nb \) less than 16, the contents of the leftmost \( nb \) bytes of VSR[XS] are placed in storage starting at address EA.

- Otherwise, the contents of VSR[XS] are placed into the quadword in storage at address EA.

Data is stored from VSR[XS] into storage in Big-Endian byte ordering (i.e., the contents of byte element 0 of VSR[XS] are placed into the byte in storage at address EA, the contents of byte element 1 of VSR[XS] are placed into the byte in storage at address EA+1, and so forth).

If the contents of bits 8:63 of GPR[RB] are not equal to 0, the results are boundedly undefined.

Special Registers Altered:

None

Example: Storing less than 16-byte left-justified data

Decimal \( X = +1234567890123456789 \);
Decimal \( Y = -123456 \);
Decimal \( Z = +1004966723510220 \);

Storing less than 16-byte data, left-justified in VSR[XS], into storage using \text{stxvll}.

VSRs X, Y, & Z

\[
\begin{array}{cccccccccc}
VSR[X]: & \text{0134 67 78 90 1234 56 78 9C} & \text{00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00} \\
VSR[Y]: & \text{01 23 45 6D} & \text{00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00} \\
VSR[Z]: & \text{01 00 49 66 72 35 10 22 0C} & \text{00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00} \\
\end{array}
\]

Initial state of Big-endian & Little-Endian storage image of X, Y, & Z

\[
\begin{array}{cccccccccccccccccccc}
X+0x0000: & \text{00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00} & \text{0134 67 78 90 1234 56 78 9C} \\
X+0x0010: & \text{01 23 45 6D} & \text{00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00} \\
\end{array}
\]

Final state of Big-endian & Little-Endian storage image of X, Y, & Z

\[
\begin{array}{cccccccccccccccccccc}
X+0x0000: & \text{01 34 67 78 90 1234 56 78 9C} & \text{01 23 45 6D} & \text{01 00} \\
X+0x0010: & \text{49 66 72 35 10 22 0C} & \text{00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00} \\
\end{array}
\]

VSR Data Layout for stxvll

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
<td>127</td>
</tr>
</tbody>
</table>

650 Power ISA™ I
Store VSX Vector Rightmost Byte Indexed X-form

```
stxvrbx  XS,RA,RB

if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB];
MEM(EA,1) = VSR[32×SX+S].byte[15];
```

Let \( XS \) be the value of \( 32 \times SX + S \).

Let \( EA \) be the sum of \( GPR[RA] \), or 0 if \( RA=0 \), and \( GPR[RB] \).

The contents of byte element 15 of \( VSR[XS] \) are placed into storage at address \( EA \).

Special Registers Altered

None

Store VSX Vector Rightmost Doubleword Indexed X-form

```
stxvrdx  XS,RA,RB

if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB];
MEM(EA,8) = VSR[32×SX+S].dword[1]
```

Let \( XS \) be the value of \( 32 \times SX + S \).

Let \( EA \) be the sum of \( GPR[RA] \), or 0 if \( RA=0 \), and \( GPR[RB] \).

Let \( store_data \) be the contents of doubleword element 1 of \( VSR[XS] \).

When Big-Endian byte ordering is employed, \( store_data \) is placed into the doubleword in storage at address \( EA \) in such an order that;

- byte 0 of \( store_data \) is placed into the byte in storage at address \( EA \),
- byte 1 of \( store_data \) is placed into the byte in storage at address \( EA+1 \), and so forth until
- byte 7 of \( store_data \) is placed into the byte in storage at address \( EA+7 \).

When Little-Endian byte ordering is employed, \( store_data \) is placed into the doubleword in storage at address \( EA \) in such an order that;

- byte 7 of \( store_data \) is placed into the byte in storage at address \( EA \),
- byte 6 of \( store_data \) is placed into the byte in storage at address \( EA+1 \), and so forth until
- byte 0 of \( store_data \) is placed into the byte in storage at address \( EA+7 \).

Special Registers Altered

None
Store VSX Vector Rightmost Halfword Indexed X-form

```
stxvrhx  XS,RA,RB
```

if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
MEM(EA,2) = VSR[32×SX+S].hword[7]

Let XS be the value of 32×SX + S.
Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].
Let store_data be the contents of halfword element 7 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed into the halfword in storage at address EA in such an order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1.

When Little-Endian byte ordering is employed, store_data is placed into the halfword in storage at address EA in such an order that;

- byte 1 of store_data is placed into the byte in storage at address EA,
- byte 0 of store_data is placed into the byte in storage at address EA+1.

Special Registers Altered
None

VSR Data Layout for stxvrhx

```
src 11 16 21 31
0 6 11 16 21 31
unused
```

Store VSX Vector Rightmost Word Indexed X-form

```
stxvwx  XS,RA,RB
```

if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
MEM(EA,4) = VSR[32×SX+S].word[3]

Let XS be the value of 32×SX + S.
Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].
Let store_data be the contents of word element 3 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed into the word in storage at address EA in such an order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 3 of store_data is placed into the byte in storage at address EA+3.

When Little-Endian byte ordering is employed, store_data is placed into the word in storage at address EA in such an order that;

- byte 3 of store_data is placed into the byte in storage at address EA,
- byte 2 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 0 of store_data is placed into the byte in storage at address EA+3.

Special Registers Altered
None

VSR Data Layout for stxvwx

```
src
0 6 11 16 21 31
unused
```

VSR[XS].word[3]

VSR[XS].hword[7]
Store VSX Vector Word*4 Indexed X-form

\[ \text{stxvw4x, XS, RA, RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>908</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

\[
\begin{align*}
\text{EA} & \leftarrow ((\text{RA}=0) \ ? \ 0 : \text{GPR}[\text{RA}]) + \text{GPR}[\text{RB}] \\
\text{MEM}(\text{EA},4) & \leftarrow \text{VSR}[32\times\text{XS}].\text{word}[0] \\
\text{MEM}(\text{EA}+4,4) & \leftarrow \text{VSR}[32\times\text{XS}].\text{word}[1] \\
\text{MEM}(\text{EA}+8,4) & \leftarrow \text{VSR}[32\times\text{XS}].\text{word}[2] \\
\text{MEM}(\text{EA}+12,4) & \leftarrow \text{VSR}[32\times\text{XS}].\text{word}[3]
\end{align*}
\]

Let \( XS \) be the value \( 32 \times \text{SX} + S \).

Let \( EA \) be the sum of the contents of \( \text{GPR}[\text{RA}] \), or 0 if \( RA \) is equal to 0, and the contents of \( \text{GPR}[\text{RB}] \).

For each integer value \( i \) from 0 to 3, do the following.

Let \( \text{store_data} \) be the contents of word element \( i \) of \( \text{VSR}[\text{XS}] \).

When Big-Endian byte ordering is employed, \( \text{store_data} \) is placed in the word in storage at address \( \text{EA}+4\times i \) in such order that:

- byte 0 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+4\times i \),
- byte 1 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+4\times i +1 \), and so forth until
- byte 3 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+4\times i +3 \).

When Little-Endian byte ordering is employed, \( \text{store_data} \) is placed in the word in storage at address \( \text{EA}+4\times i \) in such order that:

- byte 0 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+4\times i +3 \),
- byte 1 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+4\times i +2 \), and so forth until
- byte 3 of \( \text{store_data} \) is placed into the byte in storage at address \( \text{EA}+4\times i \).

Special Registers Altered

None

VSR Data Layout for \( \text{stxvw4x} \)

\[ \begin{array}{cccc}
\text{src} & \text{VSR}[\text{XS}].\text{word}[0] & \text{VSR}[\text{XS}].\text{word}[1] & \text{VSR}[\text{XS}].\text{word}[2] & \text{VSR}[\text{XS}].\text{word}[3] \\
0 & 32 & 64 & 96 & 128
\end{array} \]

Programming Note

\( \text{stxvd2x, stxvw4x, stxvh8x, stxvb16x, and stxvx} \) exhibit identical behavior in Big-Endian mode.
### Store VSX Vector Paired DQ-form

**stxvp**  
XSp,DQ(RA)

<table>
<thead>
<tr>
<th>6</th>
<th>Sp</th>
<th>31</th>
<th>RA</th>
<th>16</th>
<th>DQ</th>
<th>28</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Prefixed Store VSX Vector Paired 8LS:D-form

**pstxvp**  
XSp,D(RA),R

#### Prefix:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>//</th>
<th>R</th>
<th>//</th>
<th>d0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>6</td>
<td>9</td>
<td>14</td>
<td>12</td>
<td>d1</td>
<td>31</td>
</tr>
</tbody>
</table>

#### Suffix:

<table>
<thead>
<tr>
<th>62</th>
<th>Sp</th>
<th>31</th>
<th>RA</th>
<th>16</th>
<th>d1</th>
<th>28</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td>16</td>
<td></td>
<td>28</td>
<td>31</td>
</tr>
</tbody>
</table>

```plaintext
if MSR.VSX=0 then VSX_Unavailable();

EAbase ← (RA=0) ? 0 : GPR[RA]
if "stxvp" then
    EAdisp ← EXT64(DQ || 0b0000)
if "pstxvp" then
    EAdisp ← EXT64(d0 || d1)

if "stxvp" then EA ← EAbase + EAdisp
if "pstxvp" & R=0 then EA ← EAbase + EAdisp
if "pstxvp" & R=1 then EA ← CIA + EAdisp

store_data.bit[0:127] ← VSR[32×SX+2×Sp]
store_data.bit[128:255] ← VSR[32×SX+2×Sp+1]
MEM(EA,32) ← store_data
```

Let **XSp** be the value 32×SX + 2×Sp (i.e., only even values of XSp can be encoded in the instruction).

For **stxvp**, let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0 and the value DQ || 0b0000.

For **pstxvp**, if R=0, let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0, and the value d0 || d1, sign-extended to 64 bits.

For **pstxvp**, if R=1, let the effective address (EA) be the sum of the address of the instruction and the value d0 || d1, sign-extended to 64 bits.

Let **store_data** be the contents of VSR[XSp] concatenated with VSR[XSp+1].

When Big-Endian byte ordering is employed, **store_data** is placed into the octword in storage at address EA in such an order that;

- byte 0 of **store_data** is placed into the byte in storage at address EA,
- byte 1 of **store_data** is placed into the byte in storage at address EA+1, and so forth until
- byte 31 of **store_data** is placed into the byte in storage at address EA+31.

When Little-Endian byte ordering is employed, **store_data** is placed into the octword in storage at address EA in such an order that;

- byte 0 of **store_data** is placed into the byte in storage at address EA+31,
- byte 1 of **store_data** is placed into the byte in storage at address EA+30, and so forth until
- byte 31 of **store_data** is placed into the byte in storage at address EA.

For **pstxvp**, if R is equal to 1 and RA is not equal to 0, the instruction form is invalid.

### Special Registers Altered

None

#### Extended Mnemonics:

Extended mnemonics for **Store VSX Vector Paired Prefixed**:

<table>
<thead>
<tr>
<th>Extended mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>stxvp Rx,value(Ry)</td>
<td>pstxvp Rx,value(Ry),0</td>
</tr>
<tr>
<td>stxvp Rx,value</td>
<td>pstxvp Rx,value(0),1</td>
</tr>
</tbody>
</table>

---

### Programming Note

For best performance, EA should be word-aligned.
**Store VSX Vector Paired Indexed X-form**

`stxvpx XSp,RA,RB`

<table>
<thead>
<tr>
<th>31</th>
<th>Sp</th>
<th>SX</th>
<th>RA</th>
<th>RB</th>
<th>461</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

- if MSR.VSX=0 then VSX_Unavailable()
- \( EA \leftarrow (\text{RA}=0 \ ? \ 0 : \text{GPR}[\text{RA}]) + \text{GPR}[\text{RB}] \)
- \( \text{store\_data}.\text{bit}[0:127] \leftarrow \text{VSR}[32\times \text{SX} + 2\times \text{Sp}] \)
- \( \text{store\_data}.\text{bit}[128:255] \leftarrow \text{VSR}[32\times \text{SX} + 2\times \text{Sp}+1] \)
- \( \text{MEM}(\text{EA},32) \leftarrow \text{store\_data} \)

Let \( XSp \) be the value \( 32\times \text{SX} + 2\times \text{Sp} \) (i.e., only even values of \( XSp \) can be encoded in the instruction).

Let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0, and the integer value in GPR[RB].

Let \( \text{store\_data} \) be the contents of \( \text{VSR}[XSp] \) concatenated with \( \text{VSR}[XSp+1] \).

When Big-Endian byte ordering is employed, \( \text{store\_data} \) is placed into the octword in storage at address EA in such an order that:

- byte 0 of \( \text{store\_data} \) is placed into the byte in storage at address EA,
- byte 1 of \( \text{store\_data} \) is placed into the byte in storage at address EA+1, and so forth until
- byte 31 of \( \text{store\_data} \) is placed into the byte in storage at address EA+31.

**VSR Data Layout for stxvpx**

When Little-Endian byte ordering is employed, \( \text{store\_data} \) is placed into the octword in storage at address EA in such an order that:

- byte 0 of \( \text{store\_data} \) is placed into the byte in storage at address EA+31,
- byte 1 of \( \text{store\_data} \) is placed into the byte in storage at address EA+30, and so forth until
- byte 31 of \( \text{store\_data} \) is placed into the byte in storage at address EA.

**Special Registers Altered:**

None

**Programming Note**

For best performance, EA should be word-aligned.
Store VSX Vector Indexed X-form

```
stxvx XS,RA,RB
```

```
<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>S</th>
<th>RA</th>
<th>RB</th>
<th>396</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>
```

- If $SX=0$ & MSR.VSX=0 then VSXUnavailable()
- If $SX=1$ & MSR.VEC=0 then VectorUnavailable()

```
EA ← (RA=0) 0 : GPR[RA] + GPR[RB]
MEM[EA,16] ← VSR[32×SX+S]
```

Let $XS$ be the value $32×SX + S$.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, `store_data` is placed into the quadword in storage at address EA in such an order that;

- byte 0 of `store_data` is placed into the byte in storage at address EA,
- byte 1 of `store_data` is placed into the byte in storage at address EA+1, and so forth until
- byte 15 of `store_data` is placed into the byte in storage at address EA+15.

When Little-Endian byte ordering is employed, `store_data` is placed into the quadword in storage at address EA in such an order that;

- byte 15 of `store_data` is placed into the byte in storage at address EA,  
- byte 14 of `store_data` is placed into the byte in storage at address EA+1, and so forth until
- byte 0 of `store_data` is placed into the byte in storage at address EA+15.

Special Registers Altered:
None

--- Programming Note ---

`stxvd2x`, `stxvw4x`, `stxvh8x`, `stxvb16x`, and `stxvx` exhibit identical behavior in Big-Endian mode.

--- VSR Data Layout for stxvx ---

```
<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>127</td>
</tr>
</tbody>
</table>
```
Example: Storing data using Store VSX Vector Indexed

```c
char W[16] = { 0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0xE0, 0xE1, 0xE2, 0xE3, 0xE4, 0xE5, 0xE6, 0xE7 };
short X[8] = { 0xF0F1, 0xF2F3, 0xF4F5, 0xF6F7, 0xE0E1, 0xE2E3, 0xE4E5, 0xE6E7 };
float Y[4] = { 0xF0F1_F2F3, 0xF4F5_F6F7, 0xE0E1_E2E3, 0xE4E5_E6E7 };
double Z[2] = { 0xF0F1_F2F3_F4F5_F6F7, 0xE0E1_E2E3_E4E5_E6E7 };
```

Storing 16 bytes of data into Big-Endian storage from VSR[X5] using `stxvx`.

Big-endian storage image of W, X, Y, & Z

<table>
<thead>
<tr>
<th>addr(W+0x0000):</th>
<th>addr(W+0x0010):</th>
<th>addr(W+0x0020):</th>
<th>addr(W+0x0030):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0123456789ABCDF</td>
<td>0123456789ABCDF</td>
<td>0123456789ABCDF</td>
<td>0123456789ABCDF</td>
</tr>
</tbody>
</table>

Little-endian storage image of W, X, Y, & Z

<table>
<thead>
<tr>
<th>addr(W+0x0000):</th>
<th>addr(W+0x0010):</th>
<th>addr(W+0x0020):</th>
<th>addr(W+0x0030):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0123456789ABCDF</td>
<td>0123456789ABCDF</td>
<td>0123456789ABCDF</td>
<td>0123456789ABCDF</td>
</tr>
</tbody>
</table>

# Assumptions
#   GPR[PW] = address of W
#   GPR[PX] = address of X = GPR[PW] + 16
#   GPR[PY] = address of Y = GPR[PW] + 32
#   GPR[PZ] = address of Z = GPR[PW] + 48

```c
stxvx xW,r0,rPW
stxvx xX,r0,rPX
stxvx xY,r0,rPY
stxvx xZ,r0,rPZ
```

# Assumptions
#   GPR[PW] = address of W
#   GPR[PX] = address of X = GPR[PW] + 16
#   GPR[PY] = address of Y = GPR[PW] + 32
#   GPR[PZ] = address of Z = GPR[PW] + 48

```c
stxvx xW,r0,rPW
stxvx xX,r0,rPX
stxvx xY,r0,rPY
stxvx xZ,r0,rPZ
```
VSX Scalar Absolute Double-Precision XX2-form

**xsabsdp** XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>11</th>
<th>B</th>
<th>345</th>
<th>(B\X)</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();

\[\text{src} \leftarrow \text{VSR}[32\times B+8].\text{dword}[0]\]
\[\text{VSR}[32\times T+T].\text{dword}[0] \leftarrow \text{bfp64\_NEGATIVE\_ABSOLUTE(src)}\]
\[\text{VSR}[32\times T+T].\text{dword}[1] \leftarrow 0x0000_0000_0000_0000\]

Let XT be the value \(32\times T + T\).
Let XB be the value \(32\times B + B\).

The absolute value of the double-precision floating-point operand in doubleword element 0 of VSR[XB] is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

**Special Registers Altered**
None

**Programming Note**
This instruction can be used to operate on a single-precision source operand.

**Programming Note**
Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

**VSR Data Layout for xsabsdp**

<table>
<thead>
<tr>
<th>src</th>
<th>(\text{VSR}[XB].\text{dword}[0])</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>(\text{VSR}[XT].\text{dword}[0])</td>
<td>(0x0000_0000_0000_0000)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src</th>
<th>(\text{VSR}[VRB+32])</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>(\text{VSR}[VRT+32])</td>
</tr>
</tbody>
</table>

VSX Scalar Absolute Quad-Precision X-form

**xsabsqp** VRT,VRB

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>0</th>
<th>VRB</th>
<th>21</th>
<th>804</th>
<th>21</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();

\[\text{VSR}[VRT+32] \leftarrow \text{bfp128\_NEGATIVE\_ABSOLUTE(VSR[VRB+32])}\]

Let XT be the value VRT + 32.
Let XB be the value VRB + 32.

The absolute value of the quad-precision floating-point value in VSR[XB] is placed into VSR[XT].

**Special Registers Altered**
None
**VSX Scalar Add Double-Precision XX3-form**

\[ \text{xsadddp} \quad \text{XT,XA,XB} \]

### Code

```c
if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()
src1 ← bfp_CONVERT_FROM_BFP64(VSR[XA].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[XB].dword[0])
v ← bfp_ADD(src1, src2)
rnd ← bfp_ROUND_TO_BFP64(0b0, FPSCR.RN, v)
result ← bfp64_CONVERT_FROM_BFP(rnd)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)
vx_flag ← vxsnan_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag
if vex_flag=0 then do
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP64(result)
FPSCR.FR ← inc_flag
FPSCR.FI ← xx_flag
end
else do
FPSCR.FR ← 0b1
FPSCR.FI ← 0b1
end
```

### Notes

- **src2** is added\(^1\) to **src1**, producing a sum having unbounded range and precision.
- The sum is normalized\(^2\).

### Actions

- See Table 61, “Actions for xsadddp,” on page 660.

### Intermediate Result

- The intermediate result is rounded to double-precision using the rounding mode specified by RN.

### Final Result


- The result is placed into doubleword element 0 of **VSR[XT]** in double-precision format.

### Contents

- The contents of doubleword element 1 of **VSR[XT]** are set to 0.

### Flags

- **FPRF** is set to the class and sign of the result. **FR** is set to indicate if the result was incremented when rounded. **FI** is set to indicate the result is inexact.

### Trap-Enabled

- If a trap-enabled invalid operation exception occurs, **VSR[XT]** and **FPRF** are not modified, and **FR** and **FI** are set to 0.

### Data Layout

- See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

### Special Registers Altered

- **FPRF** FR FI FX UX XX VXSNAN VXISI

### Programming Note

- Previous versions of the architecture allowed the contents of doubeelement 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

---

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>vxsi_flag ← 1</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← -Zero</td>
<td>v ← Resz</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>vxsi_flag ← 1</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

Explanation:

src1 The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2 The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN Default quiet NaN (0x7FF8_0000_0000_0000).
NZF Nonzero finite number.
Resz Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
A(x,y) Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
Note: If x = -y, v is considered to be an exact-zero-difference result (Resz).
Q(x) Return a QNaN with the payload of x.
v The intermediate result having unbounded significand precision and unbounded exponent range.

Table 61. Actions for xsadddp
### Table 62. Scalar Floating-Point Intermediate Result Handling

<table>
<thead>
<tr>
<th>Range of ( v )</th>
<th>Case</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>v is a QNaN</td>
<td>Special</td>
<td>( r \leftarrow r \leftarrow v )</td>
</tr>
<tr>
<td>v = -( \infty )</td>
<td>Special</td>
<td>( r \leftarrow r \leftarrow v )</td>
</tr>
<tr>
<td>(-\infty &lt; v \leq -N_{\text{max}} + 1/2 \text{ulp} )</td>
<td>Overflow</td>
<td>( q \leftarrow \text{round}(v) )</td>
</tr>
<tr>
<td>(-N_{\text{max}} + 1/2 \text{ulp} &lt; v \leq -N_{\text{max}} + 1/2 \text{ulp} )</td>
<td>Overflow</td>
<td>( q \leftarrow \text{round}(v) )</td>
</tr>
<tr>
<td>(-N_{\text{max}} + 1/2 \text{ulp} &lt; v \leq -N_{\text{max}} + 1/2 \text{ulp} )</td>
<td>Normal</td>
<td>( r \leftarrow -N_{\text{max}} )</td>
</tr>
<tr>
<td>(-N_{\text{max}} + 1/2 \text{ulp} &lt; v \leq -N_{\text{max}} + 1/2 \text{ulp} )</td>
<td>Normal</td>
<td>( r \leftarrow -N_{\text{max}} )</td>
</tr>
<tr>
<td>(-N_{\text{max}} + 1/2 \text{ulp} &lt; v \leq -N_{\text{max}} + 1/2 \text{ulp} )</td>
<td>Normal</td>
<td>( r \leftarrow -N_{\text{max}} )</td>
</tr>
<tr>
<td>v = +Zero</td>
<td>Special</td>
<td>( r \leftarrow r \leftarrow v )</td>
</tr>
<tr>
<td>v = +2Zero</td>
<td>Special</td>
<td>( r \leftarrow r \leftarrow v )</td>
</tr>
<tr>
<td>+2Zero &lt; v &lt; +N_{\text{max}}</td>
<td>Tiny</td>
<td>( q \leftarrow \text{round}(v) )</td>
</tr>
<tr>
<td>+N_{\text{max}} &lt; v &lt; +Infinity</td>
<td>Normal</td>
<td>( r \leftarrow \text{round}(v) )</td>
</tr>
<tr>
<td>+N_{\text{max}} &lt; v &lt; +N_{\text{max}} + 1/2 \text{ulp}</td>
<td>Normal</td>
<td>( r \leftarrow \text{round}(v) )</td>
</tr>
<tr>
<td>+N_{\text{max}} + 1/2 \text{ulp} &lt; v &lt; +N_{\text{max}} + 1/2 \text{ulp}</td>
<td>Overflow</td>
<td>( q \leftarrow \text{round}(v) )</td>
</tr>
<tr>
<td>+N_{\text{max}} + 1/2 \text{ulp} &lt; v &lt; +N_{\text{max}} + 1/2 \text{ulp}</td>
<td>Normal</td>
<td>( r \leftarrow \text{round}(v) )</td>
</tr>
<tr>
<td>v = +Infinity</td>
<td>Special</td>
<td>( r \leftarrow r \leftarrow v )</td>
</tr>
</tbody>
</table>

**Explanation:**

1. This situation cannot occur.
2. The precise intermediate result defined in the instruction having unbounded range and precision.
3. The significand of \( v \) is shifted right by the amount of the difference between the target rounding precision \( E_{\text{min}} \) and the unbiased exponent of \( v \). The unbiased exponent of the denormalized value is \( E_{\text{min}} \). The significand of the denormalized value has unbounded significant precision.

\[
E_{\text{min}} = \begin{cases} 
-16382 & \text{(quad-precision)} \\
-16382 & \text{(double-extended-precision)} \\
-1022 & \text{(double-precision)} \\
-1022 & \text{(single-precision)}
\end{cases}
\]

4. Operates only to add operations involving source operands having the same magnitude and different signs or subtract operations involving source operands having the same magnitude and same signs. Whether +Zero or -Zero is returned is determined by the setting of the rounding mode in \( \text{RN} \), even when the rounding mode is overidden to Round to Odd.

\[
\text{round}(x) = \begin{cases} 
+1 & \text{if } x \geq 0 \\
-1 & \text{if } x < 0
\end{cases}
\]

5. The significand of \( x \) is rounded to the target rounding precision according to the rounding mode specified in \( \text{FPSCR}. \text{RN} \). Exponent range of the rounded result is unbounded. See Section 7.3.2.6.

6. Largest (in magnitude) representable normalized number in the target rounding precision format.

\[
\text{N}_{\text{max}} = \begin{cases} 
2^{127} x 1.0000000000000000000000000000000 & \text{(quad-precision)} \\
2^{127} x 1.0000000000000000000000000000000 & \text{(double-extended-precision)} \\
2^{1023} x 1.0000000000000000000000000000000 & \text{(double-precision)} \\
2^{1023} x 1.0000000000000000000000000000000 & \text{(single-precision)}
\end{cases}
\]

7. Smallest (in magnitude) representable normalized number in the target rounding precision format.

\[
\text{N}_{\text{min}} = \begin{cases} 
2^{-127} x 1.0000000000000000000000000000000 & \text{(double-precision)} \\
2^{-127} x 1.0000000000000000000000000000000 & \text{(single-precision)}
\end{cases}
\]

8. Least significant bit in the target precision format’s significand (Unit in the Last Position).

---

Chapter 7. Vector-Scalar Extension Facility 661
### Returned Results and Status Setting

| Case | FPSCR.VE | FPSCR.UE | FPSCR.ZE | FPSCR.XE | vxsnan_flag | vximz_flag | vxisi_flag | vxidi_flag | vxzdz_flag | vxsqrt_flag | zx_flag | Is r inexact? (r \neq v) | Is q inexact? (q \neq v) | Is r incremented? (| r | > | v |) | Is q incremented? (| q | > | v |) |
|------|----------|----------|----------|----------|-------------|------------|------------|------------|------------|-------------|--------|--------------------------|--------------------------|--------------------------|--------------------------|
| Special | - - - - 0 - - - - - - - - - - 1 - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(ZX) | fx(ZX), error() |
| | - - - - 1 - - - - - - - - - - 1 - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXSQRT) | |
| | 0 - - - - - - - - - - - - - - - - 1 - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXZDZ) | |
| | 0 - - - - - - - - - - - - - - - - 1 - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXIDI) | |
| | 0 - - - - - - - - - - 0 1 - - - - - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXISI) | |
| | 0 - - - - - - - - - - 1 0 - - - - - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXSNAN) | |
| | 0 - - - - - - - - - - 1 0 - - - - - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXSNAN), fx(VXISI) | |
| | 1 - - - - - - - - - - - - - - - - 1 - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXSQRT), error() | |
| | 1 - - - - - - - - - - - - - - - - 1 - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXZDZ), error() | |
| | 1 - - - - - - - - - - - - - - - - 1 - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXIDI), error() | |
| | 1 - - - - - - - - - - - - - - - - 1 - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXISI), error() | |
| | 0 - - - - - - - - - - 0 1 - - - - - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXSNAN) | |
| | 0 - - - - - - - - - - 1 0 - - - - - - - - | T(r), class_bfp(r), fi(0), fr(0), fx(VXSNAN), fx(VXIMZ) | |
| Explanation: | - The results do not depend on this condition. |
| T(x) | Places the result into the target VSR. |
| For scalar single-precision and double-precision results |
| VSR[XT].dword[0] ← bfp64_CONVERT_FROM_BFP(r) |
| VSR[XT].dword[1] ← 0x0000_0000_0000_0000 |
| For scalar quad-precision results |
| VSR[VR(32)] ← bfp128_CONVERT_FROM_BFP(r) |
| class_bfp(x) | Sets FPSCR.FPRF to the sign and class of x. |
| FPSCR.FPRF ← fprf_CLASS_BFP32(x) | (single-precision) |
| FPSCR.FPRF ← fprf_CLASS_BFP64(x) | (double-precision) |
| FPSCR.FPRF ← fprf_CLASS_BFP128(x) | (quad-precision) |
| fx(x) | FPSCR.FX is set to 1 if FPSCR.x=0. |
| fi(x) | FPSCR.FI is set to the value x. |
| fr(x) | FPSCR.FR is set to the value x. |
| β | Wrap adjust |
| \( \beta = 2^{2192} \) | (single-precision) |
| \( \beta = 2^{21536} \) | (double-precision) |
| \( \beta = 2^{224576} \) | (quad-precision) |
| See Table 7.4.3.2, “Action for OE=1,” on page 542 for trap-enabled Overflow exceptions. |
| See Table 7.4.4.2, “Action for UE=1,” on page 548 for trap-enabled Underflow exceptions. |
| q | The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target rounding precision, unbounded exponent range. |
| r | The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target rounding precision, exponent bounded to the target rounding precision format exponent range. |
| error() | The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode. |

Table 63. VSX Scalar Floating-Point Final Result
## Returned Results and Status Setting

### Normal

<table>
<thead>
<tr>
<th>Case</th>
<th>FPSCR.VE</th>
<th>FPSCR.OE</th>
<th>FPSCR.UE</th>
<th>FPSCR.ZE</th>
<th>FPSCR.XE</th>
<th>vxsnan_flag</th>
<th>vximz_flag</th>
<th>vxisi_flag</th>
<th>vxidi_flag</th>
<th>vxzdz_flag</th>
<th>vxsqrt_flag</th>
<th>zx_flag</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

### Overflow

<table>
<thead>
<tr>
<th>Case</th>
<th>FPSCR.VE</th>
<th>FPSCR.OE</th>
<th>FPSCR.UE</th>
<th>FPSCR.ZE</th>
<th>FPSCR.XE</th>
<th>vxsnan_flag</th>
<th>vximz_flag</th>
<th>vxisi_flag</th>
<th>vxidi_flag</th>
<th>vxzdz_flag</th>
<th>vxsqrt_flag</th>
<th>zx_flag</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overflow</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

### Tiny

<table>
<thead>
<tr>
<th>Case</th>
<th>FPSCR.VE</th>
<th>FPSCR.OE</th>
<th>FPSCR.UE</th>
<th>FPSCR.ZE</th>
<th>FPSCR.XE</th>
<th>vxsnan_flag</th>
<th>vximz_flag</th>
<th>vxisi_flag</th>
<th>vxidi_flag</th>
<th>vxzdz_flag</th>
<th>vxsqrt_flag</th>
<th>zx_flag</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiny</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

### Explanation:

- The results do not depend on this condition.
- \( T(x) \) places the result into the target VSR.
  - For scalar single-precision and double-precision results
    \[ \text{VR}[\text{XT}] \text{.dword}[0] \leftarrow \text{bfpu64\_CONVERT\_FROM\_BFPU}(r) \]
    \[ \text{VR}[\text{XT}] \text{.dword}[1] \leftarrow 0x0000\_0000\_0000\_0000 \]
  - For scalar quad-precision results
    \[ \text{VR}[\text{VR}+32] \leftarrow \text{bfpu128\_CONVERT\_FROM\_BFPU}(r) \]
- \( \text{class\_bfpu}(x) \)
  - Sets \( \text{FPSCR.\_FRF} \) to the sign and class of \( x \).
  - \( \text{FPSCR.\_FRF} \leftarrow \text{fprf\_CLASS\_BFPU32}(x) \) (single-precision)
  - \( \text{FPSCR.\_FRF} \leftarrow \text{fprf\_CLASS\_BFPU64}(x) \) (double-precision)
  - \( \text{FPSCR.\_FRF} \leftarrow \text{fprf\_CLASS\_BFPU128}(x) \) (quad-precision)
- \( \text{fx}(x) \)
  - \( \text{FPSCR.\_FX} \) is set to 1 if \( \text{FPSCR.\_x}=0 \).
  - \( \text{FPSCR.\_x} \) is set to 1.
- \( \text{fi}(x) \)
  - \( \text{FPSCR.\_FR} \) is set to the value \( x \).
- \( \beta \)
  - Wrap adjust
    - \( \beta = 2\frac{1}{2} \) (single-precision)
    - \( \beta = 2\frac{3}{3} \) (double-precision)
    - \( \beta = 2\frac{4}{3} \) (quad-precision)
  - See Table 7.4.3.2, “Action for OE=1,” on page 542 for trap-enabled Overflow exceptions.
  - See Table 7.4.4.2, “Action for UE=1,” on page 548 for trap-enabled Underflow exceptions.
- \( q \)
  - The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target rounding precision, unbounded exponent range.
- \( r \)
  - The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target rounding precision, exponent bounded to the target rounding precision format exponent range.
- \( \text{error}() \)
  - The system error handler is invoked for the trap-enabled exception if \( \text{MSR.\_FE0} \) and \( \text{MSR.\_FE1} \) are set to any mode other than the ignore-exception mode.

Table 63.VSX Scalar Floating-Point Final Result (Continued)
VSX Scalar Add Single-Precision XX3-form

**xsaddsp**

**XT, XA, XB**

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>O</th>
<th>A</th>
<th>B</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable();
reset_xflags();
src1 ← bfp_CONVERT_FROM_BFP64(VSR[XA].dword[0]);
src2 ← bfp_CONVERT_FROM_BFP64(VSR[XB].dword[0]);

if vsxnan_flag=1 then SetFX(FPSCR.VXSnan);
if vxisi_flag=1 then SetFX(FPSCR.VXISI);
if ox_flag=1 then SetFX(FPSCR.OX);
if ux_flag=1 then SetFX(FPSCR.UX);
if xx_flag=1 then SetFX(FPSCR.XX);

vx_flag ← vsxnan_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
VSR[32×TX+T].dword[0] ← result64
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
FPSCR.FR ← xx_flag
end
else do
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
end

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsaddsp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is added[1] to src1, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 64, “Actions for xsaddsp,” on page 665.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.


The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

Special Registers Altered

FPRF FR FI
FX OX UX XX VXSnan VXISI
Table 64. Actions for xsaddsp

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td></td>
<td>v ← Q(src2)</td>
<td>v ← vsnан_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← vsnан_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← -Zero</td>
<td>v ← Rezd</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← vsnан_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Rezd</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← vsnан_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← vsnан_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← vsnан_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← vsnан_flag ← 1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← vsnан_flag ← 1</td>
</tr>
</tbody>
</table>

Explanation:
- src1: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- src2: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- dQNaN: Default quiet NaN (0x7FF8_0000_0000_0000).
- NZF: Nonzero finite number.
- Rezd: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- A[x,y]: Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
- Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- Q[x]: Return a QNaN with the payload of x.
- v: The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Scalar Add Quad-Precision [using round to Odd] X-form

\[\text{xsaddqp \text{ VRT,VRA,VRB} \quad (RO=0)}\]
\[\text{xsaddqpo VRT,VRA,VRB} \quad (RO=1)\]

\[
\begin{array}{cccccc}
0 & 63 & 4 & 21 & 16 & 6 \\
\text{VR} & \text{VR} & \text{VR} & \text{VR} & \text{VR} & \text{VR} \\
\end{array}
\]

if MSR.VSX=0 then VSX_Unavailable() \[\text{reset_xflags()}\]

\[\text{src1} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP128(VSR[VRA+32])}\]
\[\text{src2} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP128(VSR[VRB+32])}\]
\[x \leftarrow \text{bfp\_ADD(src1, src2)}\]
\[\text{rnd} \leftarrow \text{bfp\_ROUND\_TO\_BFP128(RO, FPSCR.RN, v)}\]
\[\text{result} \leftarrow \text{bfp128\_CONVERT\_FROM\_BFP(rnd)}\]

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxisi_flag=1  then SetFX(FPSCR.VXISI)
if ox_flag=1     then SetFX(FPSCR.OX)
if ux_flag=1     then SetFX(FPSCR.UX)
if xx_flag=1     then SetFX(FPSCR.XX)

vx_flag \leftarrow \text{vxsnan_flag} | \text{vxisi_flag}
ve_flag \leftarrow \text{FPSCR.VE} & \text{vx_flag}

if ve_flag=0 then do
\[\text{VSR[VRT+32]} \leftarrow \text{result}\]
\[\text{FPSCR.FPRF} \leftarrow \text{fprf\_CLASS\_BFP128(result)}\]
end
\[\text{FPSCR.FR} \leftarrow \text{vx_flag} & \text{inc_flag}\]
\[\text{FPSCR.FI} \leftarrow \text{vx_flag} & \text{xx_flag}\]

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 and src2 are Infinity values having opposite signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src1 and src2 are Infinity values having opposite signs, the result is the default Quiet NaN\[1\].

Otherwise, do the following.

The normalized sum of src2 added to src1 is produced with unbounded significand precision and exponent range.

See Table 65, “Actions for xaddqpo[1],” on page 667.

If the intermediate result is Tiny (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right N bits, where N is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. The intermediate result is rounded to quad-precision using the specified rounding mode.


The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

Special Registers Altered:

\[\text{FPRF FR FI VXSNAN VXISI OX UX XX}\]

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.
VSR Data Layout for xsaddqp[o]

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v ← QNaN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation:

src1: The quad-precision floating-point value in VSR[VRB+32].
src2: The quad-precision floating-point value in VSR[VRB+32].
dQNaN: Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000_0000).
NZF: Nonzero finite number.
Rezd: Exact-zero-difference result (addition of two finite numbers having same magnitude and opposite signs).
add(x,y): The floating-point value y is added to the floating-point value x. Return the normalized sum, having unbounded significand precision and exponent range.
When x = -y, v is considered to be an exact-zero-difference result (Rezd).
quiet(x): Convert x to the corresponding Quiet NaN by setting the most significant fraction bit to 1.
v: The intermediate result having unbounded significand precision and unbounded exponent range.

Table 65. Actions for xsaddqp[o]

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate difference.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
VSX Scalar Compare Exponents
Double-Precision XX3-form

\texttt{xscmpexpdp} BF,XA,XB

if MSR.VSX=0 then VSX_Unavailable();

\begin{align*}
\text{src1} & \leftarrow \text{VSR}[32\times AX + A].\text{dword}[0] \\
\text{src2} & \leftarrow \text{VSR}[32\times BX + B].\text{dword}[0] \\
\text{src1}.\text{exponent} & \leftarrow \text{EXTZ}(\text{src1}.\text{bit}[1:11]) \\
\text{src2}.\text{exponent} & \leftarrow \text{EXTZ}(\text{src2}.\text{bit}[1:11]) \\
\text{src1}.\text{fraction} & \leftarrow \text{EXTZ}(\text{src1}.\text{bit}[12:63]) \\
\text{src2}.\text{fraction} & \leftarrow \text{EXTZ}(\text{src2}.\text{bit}[12:63]) \\
\text{src1}.\text{class.NaN} & \leftarrow (\text{src1}.\text{exponent} = 2047) \& (\text{src1}.\text{fraction} \neq 0) \\
\text{src2}.\text{class.NaN} & \leftarrow (\text{src2}.\text{exponent} = 2047) \& (\text{src2}.\text{fraction} \neq 0)
\end{align*}

\begin{align*}
\text{lt}_\text{flag} & \leftarrow (\text{src1}.\text{exponent} < \text{src2}.\text{exponent}) \\
\text{gt}_\text{flag} & \leftarrow (\text{src1}.\text{exponent} > \text{src2}.\text{exponent}) \\
\text{eq}_\text{flag} & \leftarrow (\text{src1}.\text{exponent} = \text{src2}.\text{exponent}) \\
\text{uo}_\text{flag} & \leftarrow \text{src1}.\text{class.NaN} \mid \text{src2}.\text{class.NaN}
\end{align*}

\begin{align*}
\text{CR}.\text{bit}[4\times BF+30] & \leftarrow \text{FPSCR.FL} \leftarrow \text{uo}_\text{flag} \mid \text{lt}_\text{flag} \\
\text{CR}.\text{bit}[4\times BF+31] & \leftarrow \text{FPSCR.FG} \leftarrow \text{uo}_\text{flag} \mid \text{gt}_\text{flag} \\
\text{CR}.\text{bit}[4\times BF+32] & \leftarrow \text{FPSCR.FE} \leftarrow \text{uo}_\text{flag} \mid \text{eq}_\text{flag} \\
\text{CR}.\text{bit}[4\times BF+33] & \leftarrow \text{FPSCR.FU} \leftarrow \text{uo}_\text{flag}
\end{align*}

Let \( \text{XA} \) be the sum \( 32\times AX + A \).

Let \( \text{XB} \) be the sum \( 32\times BX + B \).

Let \text{src1} be the double-precision floating-point value in doubleword element 0 of \text{VSR[XA]}.

Let \text{src2} be the double-precision floating-point value in doubleword element 0 of \text{VSR[XB]}.

The exponent of \text{src1} is compared with the exponent of \text{src2}. The result of the compare is placed into FPCC and CR field BF.

\textbf{Special Registers Altered:}
- CR field BF
- FPCC

\textbf{Programming Note}

This instruction can be used to operate on single-precision source operands.

\textbf{VSR Data Layout for xscmpexpdp}

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[AX].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[BX].dword[0]</td>
<td>unused</td>
</tr>
</tbody>
</table>

668 Power ISA™ I
VSX Scalar Compare Exponents
Quad-Precision X-form

xscmpexpqp BF,VRA,VRB

if MSR.VSX=0 then VSX_Unavailable()
reset_flags()

src1 ← VSR[VRA+32]
src2 ← VSR[VRB+32]

src1.exponent ← EXTZ(src1.bit[1:15])
src2.exponent ← EXTZ(src2.bit[1:15])
src1.fraction ← EXTZ(src1.bit[16:127])
src2.fraction ← EXTZ(src2.bit[16:127])

src1.class.NaN ← (src1.exponent = 32767) &
                 (src1.fraction != 0)
src2.class.NaN ← (src2.exponent = 32767) &
                 (src2.fraction != 0)

lt_flag ← (src1.exponent < src2.exponent)
gt_flag ← (src1.exponent > src2.exponent)
eq_flag ← (src1.exponent = src2.exponent)
uo_flag ← src1.class.NaN | src2.class.NaN

CR.bit[4×BF+32] ← FPSCR.FL ← !uo_flag & lt_flag
CR.bit[4×BF+33] ← FPSCR.FG ← !uo_flag & gt_flag
CR.bit[4×BF+34] ← FPSCR.FE ← !uo_flag & eq_flag
CR.bit[4×BF+35] ← FPSCR.FU ← uo_flag

Let src1 be the floating-point value in VSR[VRA+32]
represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32]
represented in quad-precision format.

The exponent of src1 is compared with the exponent of
src2 as unsigned integer values. The result of the
compare is placed into FPCC and CR field BF.

Special Registers Altered:
CR field BF
FPCC

VSR Data Layout for xscmpexpqp

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRA+32]</td>
<td>VSR[VRB+32]</td>
</tr>
</tbody>
</table>
## VSX Scalar Compare Equal Double-Precision XX3-form

 VSX Scalar Compare Equal Double-Precision XX3-form

```
xscmpeqdp XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Let $XT$ be the value $32\times TX + T$.  
Let $XA$ be the value $32\times AX + A$.  
Let $XB$ be the value $32\times BX + B$.  

Let $src1$ be the double-precision floating-point value in doubleword 0 of $VSR[XA]$.  

Let $src2$ be the double-precision floating-point value in doubleword 0 of $VSR[XB]$.  

If $src1$ or $src2$ is a SNaN, an Invalid Operation exception occurs.  

A NaN compared to any value, including itself, compares false for the predicate, equal.  

The contents of doubleword 0 of $VSR[XT]$ are set to $0xFFFF_FFFF_FFFF_FFFF$ if $src1$ is equal to $src2$, and are set to $0x0000_0000_0000_0000$ otherwise.  

The contents of doubleword 1 of $VSR[XT]$ are set to $0x0000_0000_0000_0000$.  

If a trap-enabled Invalid Operation occurs, $VSR[XT]$ is not modified.  

### Special Registers Altered:

- FX VXSNAN

### VSR Data Layout for xscmpeqdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

Let $VSR[XT].dword[0]$ be the value $32\times TX + T$.  
Let $VSR[XT].dword[1]$ be the value $0x0000_0000_0000_0000$.  
Let MSR.VSX=0 then VSX_Unavailable().  

$src1 \leftarrow \text{bfp\_CONVER FROM\_BFP64(VSR[32\times AX+A].dword[0])}$  
$src2 \leftarrow \text{bfp\_CONVER FROM\_BFP64(VSR[32\times BX+B].dword[0])}$  

$vxxn\_flag \leftarrow src1.\text{class.} SNaN | src2.\text{class.} SNaN$  

$vex\_flag \leftarrow \text{FPSCR.VE} \& vxxn\_flag$  

if $vxxn\_flag=1$ SetFX($\text{FPSCR.VXSNAN})$  

if $vex\_flag=0$ then do  
  if $src1=src2$ then  
    $VSR[32\times TX+T].dword[0] \leftarrow 0xFFFF_FFFF_FFFF_FFFF$  
    $VSR[32\times TX+T].dword[1] \leftarrow 0x0000_0000_0000_0000$  
    end  
  else do  
    $VSR[32\times TX+T].dword[0] \leftarrow 0x0000_0000_0000_0000$  
    $VSR[32\times TX+T].dword[1] \leftarrow 0x0000_0000_0000_0000$  
    end  
  end  

if $vex\_flag=1$ SetFX($\text{FPSCR.VXSNAN})$
VSX Scalar Compare Equal Quad-Precision X-form

Let \( \text{src1} \) be the quad-precision floating-point value in \( \text{VSR}[\text{VRA}+32] \).

Let \( \text{src2} \) be the quad-precision floating-point value in \( \text{VSR}[\text{VRB}+32] \).

If \( \text{src1} \) or \( \text{src2} \) is a SNaN, an Invalid Operation exception occurs.

\( \text{src1} \) is compared to \( \text{src2} \).

A NaN compared to any value, including itself, compares false for the predicate, equal.

The contents of \( \text{VSR}[\text{VRT}+32] \) are set to all 1s if \( \text{src1} \) is equal to \( \text{src2} \), and are set to all 0s otherwise.

If a trap-enabled Invalid Operation occurs, \( \text{VSR}[\text{VRT}+32] \) is not modified.

Special Registers Altered:

\[ \text{FX VXSNAN} \]

VSR Data Layout for xscmpeqqp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT]</td>
</tr>
</tbody>
</table>

0 127
VSX Scalar Compare Greater Than or Equal Double-Precision XX3-form

xscmpgedp XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>19</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable();

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

if src1.class.SNaN=1 | src2.class.SNaN=1 then do
   vxsnan_flag ← 0b1
   if FPSCR.VE=0 then vxvc_flag ← 0b1
else
   vxvc_flag ← src1.class.QNaN | src2.class.QNaN
end

vex_flag ← FPSCR.VE & (vxsnan_flag | vxvc_flag)

if vxsnan_flag=1 SetFX(FPSCR.VXSNAN)
if vxvc_flag=1 SetFX(FPSCR.VXVC)

if vex_flag=0 then do
   if src1 >= src2 then
      VSR[32×TX+T].dword[0] ← 0xFFFF_FFFF_FFFF_FFFF
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
else do
   VSR[32×TX+T].dword[0] ← 0x0000_0000_0000_0000
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
end

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].
Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].
src1 is compared to src2.

A NaN compared to any value, including itself, compares false for the predicate, greater than or equal.

The contents of doubleword 0 of VSR[XT] are set to 0xFFFF_FFFF_FFFF_FFFF if src1 is greater than or equal to src2, and are set to 0x0000_0000_0000_0000 otherwise.

The contents of doubleword 1 of VSR[XT] are set to 0x0000_0000_0000_0000.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:
FX VXSNAN VXVC

VSR Data Layout for xscmpgedp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

0 64 127
\textbf{VSX Scalar Compare Greater Than or Equal Quad-Precision X-form}

Let \( \text{src1} \) be the quad-precision floating-point value in \( \text{VSR}[\text{VRA}+32] \).

Let \( \text{src2} \) be the quad-precision floating-point value in \( \text{VSR}[\text{VRB}+32] \).

\( \text{src1} \) is compared to \( \text{src2} \).

A NaN compared to any value, including itself, compares false for the predicate, greater than or equal.

The contents of \( \text{VSR}[\text{VRT}+32] \) are set to all 1s if \( \text{src1} \) is greater than or equal to \( \text{src2} \), and are set to all 0s otherwise.

If a trap-enabled Invalid Operation occurs, \( \text{VSR}[\text{VRT}+32] \) is not modified.

\textbf{Special Registers Altered:}

\begin{itemize}
  \item FX VXSNAN VXVC
\end{itemize}
**VSX Scalar Compare Greater Than Double-Precision XX3-form**

```
xscmpgtdp  XT,XA,XB
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>11</th>
<th>MSR.VSX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSXUnavailable()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

if src1.class.SNaN=1 | src2.class.SNaN=1 then do
vxsnan_flag ← 0b1
if FPSCR.VE=0 then vxvc_flag ← 0b1
end
else
vxvc_flag ← src1.class.QNaN | src2.class.QNaN
vex_flag ← FPSCR.VE & (vxsnan_flag | vxvc_flag)
if vxsnan_flag=1 SetFX(FPSCR.VXSNAN)
if vxvc_flag=1   SetFX(FPSCR.VXVC)
if vex_flag=0 then do
  if src1 > src2 then
    VSR[32×TX+T].dword[0] ← 0xFFFF_FFFF_FFFF_FFFF
    VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
  end
else do
  VSR[32×TX+T].dword[0] ← 0x0000_0000_0000_0000
  VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
end
end
```

Let XT be the value $32 \times TX + T$.
Let XA be the value $32 \times AX + A$.
LetXB be the value $32 \times BX + B$.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

src1 is compared to src2.

A NaN compared to any value, including itself, compares false for the predicate, greater than.

The contents of doubleword 0 of VSR[VRT] are set to 0xFFFF_FFFF_FFFF_FFFF if src1 is greater than src2, and are set to 0x0000_0000_0000_0000 otherwise.

The contents of doubleword 1 of VSR[VRT] are set to 0x0000_0000_0000_0000.

If a trap-enabled Invalid Operation occurs, VSR[VRT+32] is not modified.

Special Registers Altered:
FX VXSNAN VXVC

---

**VSR Data Layout for xscmpgtdp**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

---

674  Power ISA™ I
VSX Scalar Compare Greater Than Quad-Precision X-form

Let \( \text{src1} \) be the quad-precision floating-point value in \( \text{VSR}[\text{VRA}+32] \).

Let \( \text{src2} \) be the quad-precision floating-point value in \( \text{VSR}[\text{VRB}+32] \).

\( \text{src1} \) is compared to \( \text{src2} \).

A NaN compared to any value, including itself, compares false for the predicate, greater than.

The contents of \( \text{VSR}[\text{VRT}+32] \) are set to all 1s if \( \text{src1} \) is greater than \( \text{src2} \), and are set to all 0s otherwise.

If a trap-enabled Invalid Operation occurs, \( \text{VSR}[\text{VRT}+32] \) is not modified.

**Special Registers Altered:**

\( \text{FX VXSNAN VXVC} \)

---

VSR Data Layout for \( \text{xscmpgtqp} \)

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT]</td>
</tr>
</tbody>
</table>
**VSX Scalar Compare Ordered Double-Precision XX3-form**

xscmpodp BF,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>BF</th>
<th>A</th>
<th>B</th>
<th>43</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable();
reset_flags();
src1 ← bfg_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0]);
src2 ← bfg_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0]);

if src1.class.SNaN=1 | src2.class.SNaN=1 then do
  vxsnan_flag ← 0b1
  if FPSCR.VE=0 then vxvc_flag ← 0b1
  end
else
  vxvc_flag ← src1.class.QNaN | src2.class.QNaN

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxvc_flag=1 then SetFX(FPSCR.VXVC)

CR.bit[4×BF+32] ← FPSCR.FL ← src1 < src2
CR.bit[4×BF+33] ← FPSCR.FG ← src1 > src2
CR.bit[4×BF+34] ← FPSCR.FE ← src1 = src2
CR.bit[4×BF+35] ← FPSCR.FU ← src1.class.QNaN | src1.class.SNaN | src2.class.QNaN | src2.class.SNaN

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is compared to src2.

Zeros of same or opposite signs compare equal.

Infinities of same signs compare equal.


The result of the compare is placed into CR field BF and the FPCC.

### VSR Data Layout for xscmpodp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
</tbody>
</table>

If either of the operands is a NaN, either quiet or signaling, CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, VXSNAN is set, and Invalid Operation is disabled (VE=0), VXVC is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, VXVC is set.


### Special Registers Altered

CR field BF
FPCC FX VXSNAN VXVC

### Programming Note

This instruction can be used to operate on single-precision source operands.
Table 66. Actions for xscmpodp - Part 1: Compare Ordered

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>cc=0b0010</td>
<td>cc=0b1000</td>
<td>cc=0b1000</td>
<td>cc=0b1000</td>
<td>cc=0b1000</td>
<td>cc=0b0001</td>
<td>vxvc_flag=1</td>
<td>vxsnan_flag=1</td>
<td>vxvc_flag=–00</td>
</tr>
<tr>
<td>–NZF</td>
<td>cc=0b0100</td>
<td>cc=C(src1,src2)</td>
<td>cc=0b1000</td>
<td>cc=0b1000</td>
<td>cc=0b1000</td>
<td>cc=0b0001</td>
<td>vxvc_flag=1</td>
<td>vxsnan_flag=1</td>
<td>vxvc_flag=–00</td>
</tr>
<tr>
<td>–Zero</td>
<td>cc=0b0100</td>
<td>cc=0b0100</td>
<td>cc=0b0010</td>
<td>cc=0b1000</td>
<td>cc=0b0100</td>
<td>cc=0b0001</td>
<td>vxvc_flag=1</td>
<td>vxsnan_flag=1</td>
<td>vxvc_flag=–00</td>
</tr>
<tr>
<td>+Zero</td>
<td>cc=0b0100</td>
<td>cc=0b0100</td>
<td>cc=0b0010</td>
<td>cc=0b1000</td>
<td>cc=0b0100</td>
<td>cc=0b0001</td>
<td>vxvc_flag=1</td>
<td>vxsnan_flag=1</td>
<td>vxvc_flag=–00</td>
</tr>
<tr>
<td>+NZF</td>
<td>cc=0b0100</td>
<td>cc=0b0100</td>
<td>cc=0b0010</td>
<td>cc=0b1000</td>
<td>cc=C(src1,src2)</td>
<td>cc=0b0100</td>
<td>cc=0b0001</td>
<td>vxvc_flag=1</td>
<td>vxsnan_flag=1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>cc=0b0100</td>
<td>cc=0b0100</td>
<td>cc=0b0010</td>
<td>cc=0b1000</td>
<td>cc=0b0100</td>
<td>cc=0b0001</td>
<td>vxvc_flag=1</td>
<td>vxsnan_flag=1</td>
<td>vxvc_flag=–00</td>
</tr>
<tr>
<td>QNaN</td>
<td>cc=0b0001</td>
<td>vxvc_flag=–00</td>
<td>cc=0b0001</td>
<td>vxvc_flag=–00</td>
<td>cc=0b0001</td>
<td>vxvc_flag=–00</td>
<td>cc=0b0001</td>
<td>vxvc_flag=–00</td>
<td>vxvc_flag=–00</td>
</tr>
<tr>
<td>SNaN</td>
<td>cc=0b0001</td>
<td>vxvc_flag=–00</td>
<td>cc=0b0001</td>
<td>vxvc_flag=–00</td>
<td>cc=0b0001</td>
<td>vxvc_flag=–00</td>
<td>cc=0b0001</td>
<td>vxvc_flag=–00</td>
<td>vxvc_flag=–00</td>
</tr>
</tbody>
</table>

Explanation:

- src1: The double-precision floating-point value in doubleword element 0 of VSR[XX].
- src2: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- NZF: Nonzero finite number.
- C(x,y): The floating-point value x is compared to the floating-point value y, returning one of three 4-bit results.
  - 0b0100 when x is greater than y
  - 0b0100 when x is equal to y
  - 0b0010 when x is less than y
- cc: The 4-bit result compare code.

Table 67. Actions for xscmpodp - Part 2: Result

<table>
<thead>
<tr>
<th>VE</th>
<th>vxsnan_flag</th>
<th>vxvc_flag</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>FPCC=cc, CR[BF]=–00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>FPCC=cc, CR[BF]=–00, fx(VXVC)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>FPCC=cc, CR[BF]=cc, fx(VXSNAN)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>FPCC=cc, CR[BF]=cc, fx(VXSNAN), fx(VXVC)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>FPCC=cc, CR[BF]=cc, fx(VXVC), error()</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>–</td>
<td>FPCC=cc, CR[BF]=cc, fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

Explanation:

- –: The results do not depend on this condition.
- cc: The 4-bit result as defined in Table 66.
- error(): The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
- fx[x]: FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
**VSX Scalar Compare Ordered Quad-Precision X-form**

```
xscmpoqp BF,VRA,VRB
```

```latex
\begin{array}{cccccc}
63 & 6 & BF & // & VRA & VRB & 132 \\
0 & 9 & 11 & 16 & 21 & 31
\end{array}
```

If either of the operands is a NaN, either quiet or signaling, CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, an Invalid Operation exception occurs and VXSNAN is set, and if Invalid Operation exceptions are disabled (VE=0), VXVC is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, an Invalid Operation exception occurs and VXVC is set.

**Special Registers Altered:**
- CR field BF
- FPSCR FX VXSNAN VXVC

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

src1 is compared to src2.

Zeros of same or opposite signs compare equal.
Infinities of same signs compare equal.

Bit 0 of CR field BF and FL are set to indicate if src1 is less than src2.

Bit 1 of CR field BF and FG are set to indicate if src1 is greater than src2.

Bit 2 of CR field BF and FE are set to indicate if src1 is equal to src2.

Bit 3 of CR field BF and FU are set to indicate unordered (i.e., src1 or src2 is a NaN).

**VSR Data Layout for xscmpoqp**

```
\begin{array}{c}
src1 \\
src2
\end{array}
```

\begin{array}{c}
VSR[VRA+32] \\
VSR[VRB+32]
\end{array}

678  Power ISA™ I
**VSX Scalar Compare Unordered Double-Precision XX3-form**

**xscmpudp**  
BF,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>BF</th>
<th>6</th>
<th>A</th>
<th>11</th>
<th>16</th>
<th>B</th>
<th>21</th>
<th>35</th>
<th>8B</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

vxsnan_flag ← src1.class.SNaN | src2.class.SNaN

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

CR.bit[4×BF+32] ← FPSCR.FL ← src1 < src2
CR.bit[4×BF+33] ← FPSCR.FG ← src1 > src2
CR.bit[4×BF+34] ← FPSCR.FE ← src1 = src2
CR.bit[4×BF+35] ← FPSCR.FU ← src1.class.SNaN | src1.class.QNaN | src2.class.SNaN | src2.class.QNaN

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is compared to src2.

Zeros of same or opposite signs compare equal equal.

Infinities of same signs compare equal.


The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, VXSNAN is set.


**VSR Data Layout for xscmpudp**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
</tbody>
</table>

---

**Special Registers Altered**

CR field BF  
FPCC FX VXSNAN

**Programming Note**

This instruction can be used to operate on single-precision source operands.

---

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
**Table 68. Actions for `xscmpudp` - Part 1: Compare Unordered**

<table>
<thead>
<tr>
<th>src1</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b0001</td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b0001</td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0001</td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0001</td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0001</td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b1000</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0010</td>
<td>cc ← 0b0001</td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>cc ← 0b0001</td>
<td>cc ← 0b0001</td>
<td>cc ← 0b0001</td>
<td>cc ← 0b0001</td>
<td>cc ← 0b0001</td>
<td>cc ← 0b0001</td>
<td>cc ← 0b0001</td>
<td></td>
</tr>
</tbody>
</table>
| SNaN   | vxsnan_flag ← 1
|        | vxsnan_flag ← 1
|        | vxsnan_flag ← 1
|        | vxsnan_flag ← 1
|        | vxsnan_flag ← 1
|        | vxsnan_flag ← 1
|        | vxsnan_flag ← 1

**Explanation:**

src1  The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2  The double-precision floating-point value in doubleword element 0 of VSR[XB].
NZF   Nonzero finite number.
C(x,y) The floating-point value x is compared to the floating-point value y, returning one of three 4-bit results.
0b1000 when x is greater than y
0b0100 when x is less than y
0b0010 when x is equal to y
cc    The 4-bit result compare code.

**Table 69. Actions for `xscmpudp` - Part 2: Result**

<table>
<thead>
<tr>
<th>VE</th>
<th>vxsnan_flag</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>− 0</td>
<td>FFCC=cc, CR[BF]=cc</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>FFCC=cc, CR[BF]=cc, FX(VXSNAN)</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>FFCC=cc, CR[BF]=cc, FX(VXSNAN), error()</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

−    The results do not depend on this condition.
cc  The 4-bit result as defined in Table 68.
error() The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
fx(x)  FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
VSX Scalar Compare Unordered Quad-Precision X-form

xscmpuqp BF,VRA,VRB

| 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 31 | 21 | 11 | 10 | 9 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | BF | // | VRA| VRB| 64 | // | 31|

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

src1 is compared to src2.

Zeros of same or opposite signs compare equal. Infinities of same signs compare equal.

Bit 0 of CR field BF and FL are set to indicate if src1 is less than src2.

Bit 1 of CR field BF and FG are set to indicate if src1 is greater than src2.

Bit 2 of CR field BF and FE are set to indicate if src1 is equal to src2.

Bit 3 of CR field BF and FU are set to indicate unordered (i.e., src1 or src2 is a NaN).

If either of the operands is a Signaling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

Special Registers Altered:

CR field BF
FPCC FX VXSNAN

VSR Data Layout for xscmpuqp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
</tbody>
</table>

VSR Data Layout for xscmpuqp

if MSR.VSX=0 then VSX_Unavailable();
reset_xflags();
src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

vxsnan_flag ← src1.class.SNaN | src2.class.SNaN

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

CR.bit[4×BF+32] ← FPSCR.FL ← src1 < src2
CR.bit[4×BF+33] ← FPSCR.FG ← src1 > src2
CR.bit[4×BF+34] ← FPSCR.FE ← src1 = src2
CR.bit[4×BF+35] ← FPSCR.FU ← src1.class.SNaN | src1.class.QNaN | src2.class.SNaN | src2.class.QNaN

if MSR.VSX=0 then VSX_Unavailable();
reset_xflags();
src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

vxsnan_flag ← src1.class.SNaN | src2.class.SNaN

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

CR.bit[4×BF+32] ← FPSCR.FL ← src1 < src2
CR.bit[4×BF+33] ← FPSCR.FG ← src1 > src2
CR.bit[4×BF+34] ← FPSCR.FE ← src1 = src2
CR.bit[4×BF+35] ← FPSCR.FU ← src1.class.SNaN | src1.class.QNaN | src2.class.SNaN | src2.class.QNaN
### VSX Scalar Copy Sign Double-Precision XX3-form

**指令**
xscpsgndp

<table>
<thead>
<tr>
<th>XT</th>
<th>XA</th>
<th>XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>B</td>
</tr>
<tr>
<td>176</td>
<td></td>
<td>176</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VSR</th>
<th>XT</th>
<th>XA</th>
<th>XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>62</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

#### 无载体

if MSR.VSX=0 then VSX_Unavailable()

\[
\begin{align*}
src1 & \leftarrow VSR[32 \times AX + A].dword[0] \land 0x8000_0000_0000_0000 \land 0x0000_0000_0000_0000 \\
src2 & \leftarrow VSR[32 \times BX + B].dword[0] \land 0x7FFF_FFFF_FFFF_FFFF \\
VSR[32 \times TX + T].dword[0] & \leftarrow src1 \lor src2 \\
VSR[32 \times TX + T].dword[1] & \leftarrow 0x0000_0000_0000_0000
\end{align*}
\]

Let XT be the value \(32 \times TX + T\).

LetXA be the value \(32 \times AX + A\).

Let XB be the value \(32 \times BX + B\).

Bit 0 of VSR[XT] is set to the contents of bit 0 of VSR[XA].

Bits 1:63 of VSR[XT] are set to the contents of bits 1:63 of VSR[XB].

The contents of doubleword element 1 of VSR[XT] are set to 0.

#### 特殊寄存器改变

无载体

#### 编程说明

此指令可以被用于操作单精度源数据。

#### 编程说明

前版本的架构允许结果寄存器的双字数据位1为未定义。然而，所有支持此指令的处理器都将这个结果写入双字数据位1，如要求的由此版本的架构。

### VSR Data Layout for xscpsgndp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

### VSR Data Layout for xscpsgnqp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

### VSX Scalar Copy Sign Quad-Precision X-form

**指令**
xscpsgnqp

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
<td>30</td>
</tr>
</tbody>
</table>

#### 无载体

if MSR.VSX=0 then VSX_Unavailable()

\[
\begin{align*}
src1 & \leftarrow VSR[VRA+32] \land 0x8000_0000_0000_0000_0000_0000_0000_0000 \land 0x0000_0000_0000_0000_0000_0000_0000_0000 \\
src2 & \leftarrow VSR[VRB+32] \land 0x7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF \\
VSR[VRT+32] & \leftarrow src1 \land src2
\end{align*}
\]

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

src2 is placed into VSR[VRT+32] with the sign of src1.

#### 特殊寄存器改变

无载体

### VSR Data Layout for xscpsgnqp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

682 Power ISA™ I
VSX Scalar Convert with round
Double-Precision to Half-Precision format
XX2-form

\texttt{xscvdphp} \ XT,\ XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>17</th>
<th>B</th>
<th>347</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VSX=0 then VSX_Unavailable()
reset_flags();

\texttt{src} \leftarrow \texttt{bfp\_CONVERT\_FROM\_BFP64(VSR[XB×32+B].dword[0])}
\texttt{rnd} \leftarrow \texttt{bfp\_ROUND\_TO\_BFP36(FPSCR.RN, src)}
\texttt{result} \leftarrow \texttt{bfp16\_CONVERT\_FROM\_BFP\texttt{(}rnd\texttt{)}}

if vsn_sn_flag=1 then SetFX(FPSCR.VXS\texttt{NAN})
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vex_flag \leftarrow FPSCR.VE & vsn_sn_flag

if vex_flag=0 then do
\texttt{VSR[TX×32+T].hword[0]} \leftarrow 0x0000_0000_0000
\texttt{VSR[TX×32+T].hword[1]} \leftarrow 0x0000_0000_0000_0000
\texttt{FPSCR.FPRF} \leftarrow \texttt{fprf\_CLASS\_BFP16(\texttt{result})}
end
FPSCR.FR \leftarrow (vex_flag=0) \& inc_flag
FPSCR.FI \leftarrow (vex_flag=0) \& xx_flag

Let XT be the value \(32\times TX + T\).
Let XB be the value \(32\times BX + B\).

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is an SNaN, the result is the half-precision representation of that SNaN converted to a QNaN.

Otherwise, if src is a QNaN, the result is the half-precision representation of that QNaN.

Otherwise, if src is an Infinity, the result is the half-precision representation of Infinity with the same sign as src.

Otherwise, if src is a Zero, the result is the half-precision representation of Zero with the same sign as src.

Otherwise, the result is the half-precision representation of src rounded to half-precision using the rounding mode specified by RN.

The result is zero-extended and placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in half-precision. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

**Special Registers Altered:**
- FPRF
- FR
- FI
- VXSNAN
- OX
- UX
- XX

---

**Programming Note**

This instruction can be used to operate on a single-precision source operand.

---

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
**VSX Scalar Convert Double-Precision to Quad-Precision format X-form**

xscvdpqp \(\text{VRT, VRB}\)

| 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   |    | 10 |   | 19 | 28 | 37 | 46 | 55 | 64 | 73 | 82 | 91 | 100| 109| 118| 127|   |    |    |    |    |    |    |    |    |    |    |    |    |    |

- If MSR.VSX = 0 then VSX_Unavailable();
- \(\text{src} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP64(VSR[VRB+32].dword[0])}\);
- If \(\text{src.class.SNaN}\) then
  - \(\text{result} \leftarrow \text{bfp128\_CONVERT\_FROM\_BFP(bfp\_QUIET(src))}\);
- Else
  - \(\text{result} \leftarrow \text{bfp128\_CONVERT\_FROM\_BFP(src)}\);
- \(\text{vxsnan\_flag} \leftarrow \text{src.class.SNaN}\);
- If \(\text{vxsnan\_flag}=1\) then SetX(FPSCR.VXSNAN);
- \(\text{vex\_flag} \leftarrow \text{FPSCR.VE \& vxsnan\_flag}\);
- If \(\text{vex\_flag}=0\) then do
  - \(\text{VSR[VRT+32]} \leftarrow \text{result}\);
  - \(\text{FPSCR.FPRF} \leftarrow \text{fprf\_CLASS\_BFP128(result)}\);
- \(\text{FPSCR.FR} \leftarrow 0\);
- \(\text{FPSCR.FI} \leftarrow 0\);

Let \(\text{src}\) be the floating-point value in doubleword element 0 of \(\text{VSR[VRB+32]}\) represented in double-precision format.

\(\text{src}\) is placed into \(\text{VSR[VRT+32]}\) in quad-precision format.

If \(\text{src}\) is a Signalling NaN, an Invalid Operation exception occurs and \(\text{VXSNAN}\) is set to 1.

\(\text{FPRF}\) is set to the class and sign of the result.

\(\text{FR}\) is set to 0. \(\text{FI}\) is set to 0.

If a trap-enabled Invalid Operation exception occurs, \(\text{VSR[XT]}\) and \(\text{FPRF}\) are not modified.

**Special Registers Altered:**
- \(\text{FPRF FR (set to 0)}\)
- \(\text{FI (set to 0)}\)
- \(\text{FX VXSNAN}\)

**VSR Data Layout for xscvdpqp**

- \(\text{src} \quad \text{VSR[VRB+32].dword[0]} \quad \text{unused}\)
- \(\text{tgt} \quad \text{VSR[VRT+32]}\)
VSX Scalar Convert with round
Double-Precision to Single-Precision format
XX2-form

\textbf{xscvdpsp} \ XT,\ XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>265</th>
<th>B/</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>60</td>
<td>31</td>
</tr>
</tbody>
</table>

\textbf{Programming Note}

This instruction can be used to operate on a single-precision source operand.

\textbf{Programming Note}

Previous versions of the architecture allowed the contents of words 1, 2, and 3 of the result register to be undefined, however, all processors that support this instruction write the result into both words 0 and 1 of the result register, as is required by this version of the architecture.

\textbf{Programming Note}

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

Let $XT$ be the value $32 \times TX + T$.
Let $XB$ be the value $32 \times BX + B$.

Let $src$ be the double-precision floating-point value in doubleword element 0 of $VSR[XB]$.

If $src$ is a SNaN, the result is $src$ converted to a QNaN (i.e., bit 12 of $src$ is set to 1). $VXSNAN$ is set to 1.

Otherwise, if $src$ is a QNaN, an Infinity, or a Zero, the result is $src$.

Otherwise, the result is $src$ rounded to single-precision using the rounding mode specified by RN.


The result is placed into word elements 0 and 1 of $VSR[XT]$ in single-precision format.

The contents of word elements 2 and 3 of $VSR[XT]$ are set to 0.

$FPRF$ is set to the class and sign of the result. $FR$ is set to indicate if the result was incremented when rounded. $FI$ is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, $VSR[XT]$ and $FPRF$ are not modified, and $FR$ and $FI$ are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

\textbf{Special Registers Altered}

$FPRF$ $FR$ $FI$ $FX$ $OX$ $UX$ $XX$ $VXSNAN$

\textbf{VSR Data Layout for xscvdpsp}

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32].word[0]</td>
<td>VSR[VRT+32].word[1]</td>
</tr>
</tbody>
</table>
**VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form**

```
xscvdpspn XT,XB
```

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th></th>
<th></th>
<th>B</th>
<th>267</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

- **if MSR.VSX=0 then VSX_Unavailable()**
- **reset_xflags()**

```
src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
result ← bfp32_CONVERT_FROM_BFP(src)
VSR[32×TX+T].word[0] ← result
VSR[32×TX+T].word[1] ← result
VSR[32×TX+T].word[2] ← 0x0000_0000
VSR[32×TX+T].word[3] ← 0x0000_0000
```

Let **XT** be the value \(32\times TX + T\).
Let **XB** be the value \(32\times BX + B\).

Let **src** be the single-precision floating-point value in doubleword element 0 of VSR[**XB**] represented in double-precision format.

**src** is placed into word elements 0 and 1 of VSR[**XT**] in single-precision format.

The contents of word elements 2 and 3 of VSR[**XT**] are set to 0.

**Special Registers Altered**
None

**Programming Note**
If **x** is not representable in single-precision, some exponent and/or significand bits will be discarded, likely producing undesirable results. The low-order 29 bits of the significand of **x** are discarded, more if the unbiased exponent of **x** is less than -126 (i.e., denormal). Finite values of **x** having an unbiased exponent less than -150 will return a result of Zero. Finite values of **x** having an unbiased exponent greater than 127 will result in discarding significant bits of the exponent. SNaN inputs having no significant bits in the upper 23 bits of the significand will return Infinity as the result. No status is set for any of these cases.

**Programming Note**
**xscvdpsp** should be used to convert a scalar double-precision value to vector single-precision format.
**xscvdpspn** should be used to convert a scalar single-precision value to vector single-precision format.

**Programming Note**
See the Programming Note for the **xscvdpsp** instruction.

**Programming Note**
Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

**VSR Data Layout for xscvdpspn**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[<strong>XB</strong>].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[<strong>XT</strong>].word[0]</th>
<th>VSR[<strong>XT</strong>].word[1]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>
**VSX Scalar Convert with round to zero**  
**Double-Precision to Signed Doubleword format XX2-form**

xscvdpsxds XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>//</th>
<th>B</th>
<th>344</th>
<th>30X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31X</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
rd ← bfp_ROUND_TO_INTEGER_TRUNC(src)
result ← s64_CONVERT_FROM_BFP(rnd)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
if xx_flag=1 then SetFX(FPSCR.XX)
vx_flag ← vxsnan_flag | vxcvi_flag
vex_flag ← FPSCR.VE & vx_flag
if vex_flag=0 then do
VSR[32×TX+T].dword[1] ← result
VSR[32×TX+T].dword[2] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← 0bUUUUU
FPSCR.FR ← inc_flag
FPSCR.FI ← xx_flag
end
else do
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
end

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000_0000_0000 and VXCVI is set to 1. If src is an NaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^63-1, the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^63, the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

If a trap-enabled invalid operation exception occurs,
- VSR[XT] and FPRF are not modified
- FA and FI are set to 0.

Otherwise,
- The result is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are set to 0.
- FPRF is set to an undefined value.
- FA is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

See Table 70.

**Special Registers Altered**

FPRF←0bUUUUU FR FI FX XX VXSNAN VXCVI

---

**Programming Note**

This instruction can be used to operate on a single-precision source operand.

---

**Programming Note**

xscvdpsxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a Round to Double-Precision Integer instruction that corresponds to the desired rounding mode, including xsrdpic which uses the rounding mode specified by RN.

---

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>FSRCVE</th>
<th>FSRCXE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(0), fr(0), fi(0), fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nmin-1 &lt; src &lt; Nmin</th>
<th>-</th>
<th>yes</th>
<th>T(Nmin), fr(0), fi(0), fx(XA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yes</td>
<td></td>
<td>T(Nmin), fr(0), fi(0), fx(XA), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src = Nmin</th>
<th>-</th>
<th>-</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T(Nmin), fr(0), fi(0), fx(XA), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nmin &lt; src &lt; Nmax</th>
<th>-</th>
<th>yes</th>
<th>T(Nmax), fr(0), fi(0), fx(XA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yes</td>
<td></td>
<td>T(Nmax), fr(0), fi(0), fx(XA), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src = Nmax</th>
<th>-</th>
<th>-</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T(Nmax), fr(0), fi(0), fx(XA), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nmax &lt; src &lt; Nmax+1</th>
<th>-</th>
<th>yes</th>
<th>T(Nmax), fr(0), fi(0), fx(XA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yes</td>
<td></td>
<td>T(Nmax), fr(0), fi(0), fx(XA), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src ≥ Nmax+1</th>
<th>-</th>
<th>-</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T(Nmax), fr(0), fi(0), fx(VXCVI), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src is a QNaN</th>
<th>-</th>
<th>-</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src is a SNaN</th>
<th>-</th>
<th>-</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T(0), fr(0), fi(0)</td>
</tr>
</tbody>
</table>

Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.

**Explanation:**

- `error()` The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode.
- `f2i(x)` The double-precision floating-point integer value x is converted to 64-bit signed integer format.
- `fi(x)` FPSCR.FI is set to the value x.
- `fr(x)` FPSCR.FR is set to the value x.
- `fx(x)` FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
- `Nmin` The smallest signed integer doubleword value, $-2^{63}$ (0x8000_0000_0000_0000).
- `Nmax` The largest signed integer doubleword value, $2^{63}-1$ (0x7FFF_FFFF_FFFF_FFFF).
- `src` The double-precision floating-point value in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are set to 0.
- `trunc(x)` The double-precision floating-point value x is truncated to a floating-point integer.
- `T(x)` The signed integer doubleword value x is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are set to 0.

Table 70. Actions for xscvdpsxd

---

Length: 688
VSX Scalar Convert with round to zero
Double-Precision to Signed Word format
XX2-form

\[
\text{xscvdpsxws} \quad \text{XT, XB}
\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>H</th>
<th>B</th>
<th>88</th>
<th>R</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
rnd ← bfp_ROUND_TO_INTEGER_TRUNC(src)
result ← s:i32_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vxcvi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
VSR[32×TX+T].word[0] ← result
VSR[32×TX+T].word[1] ← result
VSR[32×TX+T].word[2] ← 0x0000_0000
VSR[32×TX+T].word[3] ← 0x0000_0000
FPSCR.FR ← 0bUUUUU
FPSCR.FR ← inc_flag
FPSCR.FI ← xx_flag
end
else do
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
end

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in
doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000 and
VXCVI is set to 1. If src is an SNaN, VXSNAN is also set
to 1.

Otherwise, src is rounded to a floating-point integer
using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{31}, the result is
0x7FFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than \(-2^{31}\), the
result is 0x8000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted
to 32-bit signed-integer format, and if the result is inexact
(i.e., not equal to src), XX is set to 1.

If a trap-enabled invalid operation exception occurs,

- VSR[XT] and FPRF are not modified
- FR and FI are set to 0.

Otherwise,

- The result is placed into word elements 0 and 1 of
  VSR[XT]. The contents of word elements 2 and 3 of
  VSR[XT] are set to 0.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented
  when rounded.
- FI is set to indicate the result is inexact.

See Table 71.

Special Registers Altered

<table>
<thead>
<tr>
<th>FPRF=0bUUUUU</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>XX</th>
<th>VXSNAN</th>
<th>VXCVI</th>
</tr>
</thead>
</table>

Programming Note

Previous versions of the architecture allowed the
contents of word 0 of the result register to be unde-
fined. However, all processors that support this
instruction write the result into words 0 and 1 of
the result register, as is required by this version of the
architecture.

Programming Note

Previous versions of the architecture allowed the
contents of doubleword 1 of the result register to be
undefined. However, all processors that support
this instruction write 0s into doubleword 1 of
the result register, as is required by this version of the
architecture.

Programming Note

This instruction can be used to operate on a
single-precision source operand.

Programming Note

xscvdpsxws rounds using Round towards Zero
rounding mode. For other rounding modes, software
must use a Round to Double-Precision
Integer instruction that corresponds to the desired
rounding mode, including xsrdpic which uses the
rounding mode specified by RN.
VSR Data Layout for xscvdpsxws

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XT].word[0]</td>
<td>VSR[XT].word[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returned Results and Status Setting</th>
<th>FPSCR.FR</th>
<th>FPSCR.FI</th>
<th>FPSCR.FE</th>
<th>(trunc(0b001, src) != src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>src ≤ Nmin + 1</td>
<td>0 -</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 -</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(1), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>Nmin + 1 &lt; src &lt; Nmax</td>
<td>0 no</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(1), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 yes</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(1), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src = Nmin</td>
<td>- no</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0)</td>
<td></td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax + 1</td>
<td>0 no</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(1), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 yes</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(1), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src ≥ Nmax + 1</td>
<td>0 -</td>
<td>-</td>
<td>T(Nmax), fr(0), fi(0), fx(VXCVI)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 -</td>
<td>-</td>
<td>T(Nmax), fr(0), fi(1), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0 -</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXCVI)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 -</td>
<td>-</td>
<td>T(Nmax), fr(0), fi(1), fx(VXCVI), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0 -</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXCVI), fx(VXSNAN)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 -</td>
<td>-</td>
<td>T(Nmax), fr(0), fi(1), fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
</tr>
</tbody>
</table>

Explanation:

- **error()** The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
- **f2i(x)** The double-precision floating-point integer value x is converted to 32-bit signed integer format.
- **fr(x)** FPSCR.FR is set to the value x.
- **fi(x)** FPSCR.FI is set to the value x.
- **fx(x)** FPSCR.FX is set to 1 if FPSCR.x=1. FPSCR.x is set to 1.
- **Nmin** The smallest signed integer word value, -2^31 (0x8000_0000).
- **Nmax** The largest signed integer word value, 2^31 -1 (0x7FFFFFFF).
- **src** The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **trunc(x)** The double-precision floating-point value x is truncated to a floating-point integer.
- **T(x)** The signed integer word value x is placed in word elements 0 and 1 of VSR[XT]. The contents of word elements 2 and 3 of VSR[XT] are set to 0.

Table 71. Actions for xscvdpsxws
VSX Scalar Convert with round to zero
Double-Precision to Unsigned Doubleword format XX2-form

`xscvdpuxds XT,XB`

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>I</th>
<th>H</th>
<th>B</th>
<th>32B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>32B</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

`src ← bfp_CONVERT_FROM_BFP64(VSR[XB].dword[0])`

`rnd ← bfp_ROUND_TO_INTEGER_TRUNC(src)`

`result ← ui64_CONVERT_FROM_BFP(rnd)`

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)

if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vxcvi_flag

vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do

VSR[32×TX+T].dword[1] ← result

VSR[32×TX+T].dword[2] ← 0x0000_0000_0000_0000

FPSCR.FPRF ← 0bUUUUU

FPSCR.FR ← inc_flag

FPSCR.FI ← xx_flag

end

else do

FPSCR.FR ← 0b0

FPSCR.FI ← 0b0

end

Let XT be the value 32×TX + T.

Let XB be the value 32×BX + B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If `src` is a NaN, the result is the value 0x0000_0000_0000_0000 and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^64 - 1, the result is 0xFFFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format, and if the result is inexact (i.e., not equal to `src`), XX is set to 1.

If a trap-enabled invalid operation exception occurs,

– VSX[XT] and FPRF are not modified
– FR and FI are set to 0.

Otherwise,

– The result is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are set to 0.
– FPRF is set to an undefined value.
– FR is set to indicate if the result was incremented when rounded.
– FI is set to indicate the result is inexact.

See Table 72.

Special Registers Altered

FPRF=0bUUUUU FR FI FX XX VXSNAN VXCVI

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

`xscvdpuxds` rounds using Round towards Zero rounding mode. For other rounding modes, software must use a Round to Double-Precision Integer instruction that corresponds to the desired rounding mode, including `xsrdpic` which uses the rounding mode specified by RN.

Programming Note

This instruction can be used to operate on a single-precision source operand.
Table 72. Actions for xscvdpuxds

<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>FPSCR.VE</th>
<th>FPSCR.IE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

| Nmin-1 < src < Nmin | 0 | yes | T(Nmin), fr(0), fi(0), fx(XAX) |
|                    | 1 | yes | T(Nmin), fr(0), fr(1), fx(XAX), error() |

| src = Nmin | - | - | no | T(Nmin), fr(0), fi(0) |

| Nmin < src < Nmax | - | - | yes | T(Nmin+1), fr(0), fi(0), fx(XAX) |
|                  | 1 | yes | T(Nmin+1), fr(0), fr(1), fx(XAX), error() |

| src = Nmax | - | - | no | T(Nmax), fr(0), fi(0) |

| Nmax < src < Nmax+1 | - | - | yes | T(Nmax), fr(0), fi(0), fx(XAX) |
|                    | 1 | yes | T(Nmax), fr(0), fr(1), fx(XAX), error() |

| src ≥ Nmax+1 | 0 | - | - | T(Nmax), fr(0), fi(0), fx(VXCVI) |
|              | 1 | - | - | T(Nmax), fr(0), fr(1), fx(VXCVI), error() |

| src is a QNaN | 0 | - | - | T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN) |
|              | 1 | - | - | T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error() |

| src is a SNaN | 0 | - | - | T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN) |
|              | 1 | - | - | T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error() |

**Explanation:**

- **error()**  
  The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.

- **f2i(x)**  
  The double-precision floating-point integer value x is converted to a 64-bit unsigned integer format.

- **fi(x)**  
  FPSCR.FI is set to the value x.

- **fr(x)**  
  FPSCR.FR is set to the value x.

- **fx(x)**  
  FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.

- **Nmin**  
  The smallest unsigned integer doubleword value, \(0x0000_0000_0000_0000\).

- **Nmax**  
  The largest unsigned integer doubleword value, \(2^{64}-1\) (\(0x{FFFF_FFFF_FFFF_FFFF}\)).

- **src**  
  The double-precision floating-point value in doubleword element 0 of VSR[XB].

- **trunc(x)**  
  The double-precision floating-point value x is truncated to a floating-point integer.

- **T(x)**  
  The unsigned integer doubleword value x is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are set to 0.
VSX Scalar Convert with round to zero
Double-Precision to Unsigned Word format
XX2-form

**xscvdpuxws XT,XB**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>T</th>
<th></th>
<th></th>
<th>B</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>30</td>
<td>31</td>
<td>BIP</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags();
src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
rnd ← bfp_ROUND_TO_INTEGER_TRUNC(src)
result ← u32_CONVERT_FROM_BFP(rnd)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
if xx_flag=1 then SetFX(FPSCR.XX)
vx_flag ← vxsnan_flag | vxcvi_flag
vex_flag ← FPSCR.VE & vx_flag
if vex_flag=0 then do
VSR[32×TX+T].word[0] ← result
VSR[32×TX+T].word[1] ← result
VSR[32×TX+T].word[2] ← 0x0000_0000
VSR[32×TX+T].word[3] ← 0x0000_0000
FPSCR.FPRF ← 0bUUUUU
FPSCR.FR ← inc_flag
FPSCR.FI ← xx_flag
end
else do
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
end

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^32-1, the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

If a trap-enabled invalid operation exception occurs,
- VSR[XT] and FPRF are not modified
- FR and FI are set to 0.

Otherwise,
- The result is placed into word elements 0 and 1 of VSR[XT]. The contents of word elements 2 and 3 of VSR[XT] are set to 0.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

See Table 73.

**Special Registers Altered**

FPRF=0bUUUUU FR FI FX XX VXSNAN VXCVI

**Programming Note**

Previous versions of the architecture allowed the contents of word 0 of the result register to be undefined. However, all processors that support this instruction write the result into words 0 and 1 of the result register, as is required by this version of the architecture.

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

**Programming Note**

This instruction can be used to operate on a single-precision source operand.

**Programming Note**

xscvdpuxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a Round to Double-Precision Integer instruction that corresponds to the desired rounding mode, including xsrdpic which uses the rounding mode specified by RN.
### VSR Data Layout for xscvdpxws

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XT].word[0]</td>
<td>VSR[XT].word[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returned Results and Status Setting</th>
<th>FPRSCR.FE</th>
<th>FPRSCR.XE</th>
<th>fp2i(trunc(0b001,src)) != src</th>
</tr>
</thead>
<tbody>
<tr>
<td>src ≤ Nmin - 1</td>
<td>0</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin - 1 &lt; src &lt; Nmin</td>
<td>-</td>
<td>0</td>
<td>yes T(Nmin), fr(0), fi(0), fx(XX)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>T(Nmin), fr(0), fi(1), fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>0</td>
<td>-</td>
<td>no T(Nmin), fr(0), fi(0)</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>-</td>
<td>-</td>
<td>no T(Nmax), fr(0), fi(0)</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>0</td>
<td>-</td>
<td>yes T(Nmax), fr(0), fi(0), fx(XX)</td>
</tr>
<tr>
<td>src ≥ Nmax+1</td>
<td>0</td>
<td>-</td>
<td>yes T(Nmax), fr(0), fi(1), fx(XX), error()</td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>-</td>
<td>- T(Nmin), fr(0), fi(0), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>- T(Nmax), fr(0), fi(0), fx(VXCVI), error()</td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0</td>
<td>-</td>
<td>- T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>- T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

**Explanation:**

- **error()** The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.

- **f2i(x)** The double-precision floating-point integer value x is converted to 32-bit unsigned integer format.

- **fi(x)** FPSCR FI is set to the value x.

- **fr(x)** FPSCR FR is set to the value x.

- **fx(x)** FPSCR FX is set to 1 if FPSCR.x$\oplus$1. FPSCR.x is set to 1.

- **Nmin** The smallest unsigned integer word value, 0 (0x0000_0000).

- **Nmax** The largest unsigned integer word value, $2^{32} - 1$ (0xFFFF_FFFF).

- **src** The double-precision floating-point value x is truncated to a floating-point integer.

- **T(x)** The unsigned integer word value x is placed in word elements 0 and 1 of VSR[XT].

The contents of word elements 2 and 3 of VSR[XT] are set to 0.

### Table 73. Actions for xscvdpxws
VSX Scalar Convert Half-Precision to Double-Precision format XX2-form

```plaintext
xscvhpdp XT,XB

if MSR.VSX=0 then VSX_Unavailable()
reset_flags()

src ← bfp_CONVERT_FROM_BFP16(VSR[BX×32+B].hword[3])
if src.class.SNaN=1 then
result ← bfp64_CONVERT_FROM_BFP(bfp_QUIET(src))
else
result ← bfp64_CONVERT_FROM_BFP(src)

vxsnan_flag ← src.class.SNaN
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag
if vex_flag=0 then do
VSR[TX×32+T].dword[0] ← result
VSR[TX×32+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← 0
FPSCR.FI ← 0

Let XT be the value \(32\times TX + T\).
Let XB be the value \(32\times BX + B\).
```

Let \(src\) be the half-precision floating-point value in the rightmost halfword of doubleword element 0 of VSR[XB].

If \(src\) is an SNaN, the result is the double-precision representation of that SNaN converted to a QNaN.

Otherwise, if \(src\) is a QNaN, the result is the double-precision representation of that QNaN.

Otherwise, if \(src\) is an Infinity, the result is the double-precision representation of Infinity with the same sign as \(src\).

Otherwise, if \(src\) is a Zero, the result is the double-precision representation of Zero with the same sign as \(src\).

Otherwise, if \(src\) is a denormal value, the result is the normalized double-precision representation of \(src\).

Otherwise, the result is the double-precision representation of \(src\).

The result is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in double-precision format.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified.

FR is set to 0. FI is set to 0.

Special Registers Altered:

- **FPRF** (set to 0)
- **FR** (set to 0)
- **FI** (set to 0)
- **FX** VXSNAN

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xscvhpdp

<table>
<thead>
<tr>
<th>src</th>
<th>unused</th>
<th>VSR[XT].hword[3]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
<td>127</td>
</tr>
</tbody>
</table>

Chapter 7. Vector-Scalar Extension Facility  695
VSX Scalar Convert with round
Quad-Precision to Double-Precision format
(\textit{using round to Odd} \ X\text{-form})

\begin{verbatim}
xscvqpdp  VRT,VRB  (RO=0)
xscvqdp  VRT,VRB  (RO=1)
\end{verbatim}

\begin{verbatim}
if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()
src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
rnd ← bfp_ROUND_TO_BFP64(RO,FPSCR.RN,src)
result ← bfp64_CONVERT_FROM_BFP(rnd)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)
vex_flag ← FPSCR.VE & vxsnan_flag
if vex_flag=0 then do
  VSR[VRT+32].dword[0] ← result
  VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
  FPSCR.FPRF ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← (vxsnan_flag=0) & inc_flag
FPSCR.FI ← (vxsnan_flag=0) & xx_flag
\end{verbatim}

Let \( src \) be the quad-precision floating-point value in \( VSR[VRB+32] \).

If \( src \) is a Signalling NaN, an Invalid Operation exception occurs and \( VXSNAN \) is set to 1.

If \( src \) is a Signalling NaN, the result is the Quiet NaN corresponding to the Signalling NaN, with the significand truncated to the rounding precision.

Otherwise, if \( src \) is a Quiet NaN, then the result is \( src \) with the significand truncated to the double-precision.

Otherwise, if \( src \) is an Infinity or a Zero, the result is \( src \).

The result is placed into doubleword element 0 of \( VSR[VRT+32] \) in double-precision format. The contents of doubleword element 1 of \( VSR[VRT+32] \) are set to 0.

\( FPRF \) is set to the class and sign of the result as represented in double-precision format. \( FR \) is set to indicate if the rounded result was incremented. \( FI \) is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, \( FR \) and \( FI \) are set to 0.

If a trap-enabled Invalid Operation exception occurs, \( VSR[VRT+32] \) and \( FPRF \) are not modified, and \( FR \) and \( FI \) are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

Special Registers Altered:
\( FPRF \ FR \ FI \ FX \ VXSNAN \ OX \ UX \ XX \)
Let `src` be the quad-precision floating-point value in `VSR[VRB+32]`.

If `src` is a Signalling NaN, an Invalid Operation exception occurs and `VXSNAN` and `VXCVI` are set to 1.

If `src` is a Quiet NaN or an Infinity, an Invalid Operation exception occurs and `VXCVI` is set to 1.

If `src` is a NaN, the result is `0x8000_0000_0000_0000`.

Otherwise, if `src` is a Zero, the result is `0x0000_0000_0000_0000`.

Otherwise, if `src` is `-Infinity`, the result is `0x7FFF_FFFF_FFFF_FFFF`.

Otherwise, if `src` is `-Infinity`, the result is `0x8000_0000_0000_0000`.

Otherwise, do the following.

Let `rnd` be the value `src` truncated to a floating-point integer.

If `rnd` is greater than `+2^{63}-1`, an Invalid Operation exception occurs, `VXCVI` is set to 1, and the result is `0x7FFF_FFFF_FFFF_FFFF`.

Otherwise, if `rnd` is less than `-2^{63}`, an Invalid Operation exception occurs, `VXCVI` is set to 1, and the result is `0x8000_0000_0000_0000`.

Otherwise, the result is the value `rnd`, and an Inexact exception occurs if `rnd` is inexact (i.e., `rnd` is not equal to `src`).

The result is placed into doubleword element 0 of `VSR[VRT+32]` in signed integer format.

The contents of doubleword element 1 of `VSR[VRT+32]` are set to 0.

`FPRF` is set to undefined. `FR` is set to 0. `FI` is set to indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, `FR` and `FI` are set to 0.

If a trap-enabled Invalid Operation exception occurs, `VSR[VRT+32]` and `FPRF` are not modified.

See Table 70, “Actions for xscvdpsxds,” on page 688.

Special Registers Altered:

`FPRF` (undefined) `FR` `FI` `FX` `VXSNAN` `VXCVI` `XX`
VSR Data Layout for xscvqpsdz

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
<th>tgt</th>
<th>VSR[VRT+32].dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Returned Results and Status Setting</th>
<th>FPSCR.FE</th>
<th>FPSCR.XE</th>
<th>FPSCR.FI</th>
<th>FPSCR.FPRF</th>
<th>FPSCR.FX</th>
<th>error()</th>
</tr>
</thead>
<tbody>
<tr>
<td>src ≤ Nmin-1</td>
<td>0</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>fr(0), fi(0), fx(VXCVI)</td>
<td>error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>-</td>
<td>yes</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmin</td>
<td>-</td>
<td>no</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>-</td>
<td>no</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(XX)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmax</td>
<td>-</td>
<td>no</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>-</td>
<td>no</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(XX)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU)</td>
<td>error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(XX)</td>
<td>error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src ≥ Nmax+1</td>
<td>0</td>
<td>-</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</td>
<td>error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNaN)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNaN), error()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNaN)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNaN), error()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation:

- **T(x)**: Places the value x into the target VSR.
  
- **VSR[VRB+32], dword[0]**: x

- **trunc(x)**: Return the quad-precision floating-point value x truncated to a floating-point integer.

- **fp2i(x)**: The quad-precision floating-point integer value x is converted to 64-bit signed integer format.

- **fp(x)**: The largest signed integer doubleword value, \(2^{63} \times (0x7FFF_FFFF_FFFF_FFFF)\).

- **fp(x)**: The smallest signed integer doubleword value, \(-2^{63} \times (0x8000_0000_0000_0000)\).

- **src**: The quad-precision floating-point value in VSR[VRB+32].

- **fi(x)**: FPSCR.FI is set to the value x.

- **fr(x)**: FPSCR.FR is set to the value x.

- **fprf(x)**: FPSCR.FPRF is set to the value x.

- **fx(x)**: FPSCR.FX is set to the value x.

- **error()**: The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.

Table 74. Actions for xscvqpsdz
Let \( src \) be the quad-precision floating-point value in \( VSR[VRB+32] \).

If \( src \) is a Signalling NaN, an Invalid Operation exception occurs and \( VXSNAN \) and \( VXCVI \) are set to 1.

If \( src \) is a Quiet NaN or an Infinity, an Invalid Operation exception occurs and \( VXCVI \) is set to 1.

If \( src \) is a NaN, the result is \(-2^{127}\).

Otherwise, if \( src \) is a Zero, the result is 0.

Otherwise, if \( src \) is +Infinity, the result is \( 2^{127}-1 \).

Otherwise, if \( src \) is -Infinity, the result is \(-2^{127}\).

Otherwise, do the following.

Let \( \text{rnd} \) be the value \( src \) truncated to a floating-point integer.

If \( \text{rnd} \) is greater than \(+2^{127}-1\), an Invalid Operation exception occurs, \( VXCVI \) is set to 1, and the result is \( 2^{127} \).

Otherwise, if \( \text{rnd} \) is less than \(-2^{127}\), an Invalid Operation exception occurs, \( VXCVI \) is set to 1, and the result is \(-2^{127}\).

Otherwise, the result is the value \( \text{rnd} \), and an Inexact exception occurs if \( \text{rnd} \) is inexact (i.e., \( \text{rnd} \) is not equal to \( src \)).

The result is placed into doubleword element 0 of \( VSR[VRT+32] \) in signed integer format.

The contents of doubleword element 1 of \( VSR[VRT+32] \) are set to 0.

\( FPRF \) is set to undefined. \( FR \) is set to 0. \( FI \) is set to indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, \( FR \) and \( FI \) are set to 0.

If a trap-enabled Invalid Operation exception occurs, \( VSR[VRT+32] \) and \( FPRF \) are not modified.

See Table 75, “Actions for xscvqpsqz,” on page 700.

Special Registers Altered:

- \( FPRF \) (undefined)
- \( FR \) (set to 0)
- \( FI \)
- \( VXSNAN\)
- \( VXCVI\)
- \( XX\)

VSR Data Layout for xscvqpsqz

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
FPSCR.VE | FPSCR.XE | Returned Results and Status Setting
--- | --- | ---
src ≤ Nmin-1 | 0 | - \( T(N_{min}), f(0), f(0), fprf(0bUUUUU), fx(VXCVI) \)
 | 1 | - \( T(N_{min}), f(0), f(0), fx(VXCVI), e(0) \)
Nmin-1 < src < Nmin | - | 0 \( T(N_{min}), f(0), f(0), fprf(0bUUUUU), fx(XX), e(0) \)
 | - | 1 \( T(N_{min}), f(0), f(0), fprf(0bUUUUU), fx(XX), e(0) \)
src = Nmin | - | - \( T(N_{min}), f(0), f(0), fprf(0bUUUUU) \)
Nmin < src < Nmax | - | 0 \( T(\{2\cdot\text{trunc}(\text{src})\}), f(0), f(0), fprf(0bUUUUU), fx(XX), e(0) \)
 | - | 1 \( T(\{2\cdot\text{trunc}(\text{src})\}), f(0), f(0), fprf(0bUUUUU), fx(XX), e(0) \)
src = Nmax | - | - \( T(N_{max}), f(0), f(0), fprf(0bUUUUU) \)
Nmax < src < Nmax+1 | - | 0 \( T(N_{max}), f(0), f(0), fprf(0bUUUUU), fx(XX) \)
 | - | 1 \( T(N_{max}), f(0), f(0), fprf(0bUUUUU), fx(XX), e(0) \)
src ≥ Nmax+1 | 0 | - \( T(N_{max}), f(0), f(0), fprf(0bUUUUU), fx(VXCVI) \)
 | 1 | - \( T(N_{max}), f(0), f(0), fx(VXCVI), e(0) \)
src is a QNaN | 0 | - \( T(N_{min}), f(0), f(0), fprf(0bUUUUU), fx(VXCVI), f(0) \)
 | 1 | - \( T(N_{min}), f(0), f(0), fx(VXCVI), f(0), e(0) \)
src is a SNaN | 0 | - \( T(N_{min}), f(0), f(0), fprf(0bUUUUU), fx(VXCVI), f(0), fx(VXSNAN) \)
 | 1 | - \( T(N_{min}), f(0), f(0), fx(VXCVI), f(0), fx(VXSNAN), e(0) \)

**Explanation:**
- \( T(x) \) Places the value \( x \) into the target VSR.
- \( N_{min} \) The smallest signed integer doubleword value, \(-2^{127} \cdot 0x8000_0000_0000_0000_0000_0000_0000_0000\).
- \( N_{max} \) The largest signed integer doubleword value, \( 2^{127-1} \cdot (0x7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF) \).
- \( src \) The quad-precision floating-point value in VSR[VRB+32].
- \( \{2\cdot\text{trunc}(x)\} \) The quad-precision floating-point integer value \( x \) is converted to 128-bit signed integer format.
- \( f(x) \) FPSCR.\( FX \) is set to 1 if FPSCR.\( x \) = 1. FPSCR.\( x \) is set to 1.
- \( f(x) \) FPSCR.\( FI \) is set to the value \( x \).
- \( f(x) \) FPSCR.\( FX \) is set to the value \( x \).
- \( fprf(x) \) FPSCR.\( FPRF \) is set to the value \( x \).
- \( error() \) The system error handler is invoked for the trap-enabled exception if MSR.\( FE0 \) and MSR.\( FE1 \) are set to any mode other than the ignore-exception mode.
- \( \text{trunc}(x) \) Returns the floating-point value \( x \) truncated to a floating-point integer.

Table 75. Actions for xcvqpsqz
VSX Scalar Convert with round to zero
Quad-Precision to Signed Word format X-form

xscvqpswz VRT,VRB

| 63 | 6 | VRT | 9 | VRB | 21 | 836 | 1 |

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()
src ← bfp_CONVERT_FROM_BFP128(VSR[VRT+32])
if src.class.QNaN=1 | src.class.SNaN=1 then do
  result ← 0xFFFF_FFFF_8000_0000
  vxsnan_flag ← src.class.S NaN
  vxcvl_flag ← 1
end
else if src.class.Infinity=1 then do
  vxcvl_flag ← 1
  if src.sign=0 then
    result ← 0x0000_0000_7FFF_FFFF
  else
    result ← 0xFFFF_FFFF_8000_0000
  end
else if src.class.Zero=1 then
  result ← 0x0000_0000_0000_0000
else do
  rnd ← bfp_ROUND_TO_INTEGER(0b001,src)
  if bfp_COMPARE_GT(rnd, +231-1) then do
    result ← 0x0000_0000_7FFF_FFFF
    vxcvl_flag ← 1
  end
  else if bfp_COMPARE_LT(rnd, -231) then do
    result ← 0xFFFF_FFFF_8000_0000
    vxcvl_flag ← 1
  end
  else do
    result ← si64_CONVERT_FROM_BFP(rnd)
    if xx_flag=1 then SetFX(FPSCR.XX)
  end
end
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvl_flag=1 then SetFX(FPSCR.VXCVI)
vx_flag ← vxsnan_flag | vxcvl_flag
ex_flag ← FPSCR.VE & vx_flag
if ex_flag=0 then do
  VSRI[VRT+32].dword[0] ← result
  VSRI[VRT+32].dword[1] ← 0x0000_0000_0000_0000
  FPSCR.FR ← 0bUUUUU
end
FPSCR.FR ← 0
FPSCR.FI ← (vx_flag=0) & xx_flag

Let src be the quad-precision floating-point value in VSR[VRT+32].

If src is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN and VXCVI are set to 1.

If src is a Quiet NaN or an Infinity, an Invalid Operation exception occurs and VXCVI is set to 1.

If src is a NaN, the result is 0xFFFF_FFFF_8000_0000.

Otherwise, if src is a Zero, the result is 0x0000_0000_0000_0000.

Otherwise, if src is a +Infinity, the result is 0x0000_0000_7FFF_FFFF.

Otherwise, if src is a -Infinity, the result is 0xFFFF_FFFF_8000_0000.

Otherwise, do the following.
Let rnd be the value src truncated to a floating-point integer.

If rnd is greater than +231-1, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0x0000_0000_7FFF_FFFF.

Otherwise, if rnd is less than -231, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0xFFFF_FFFF_8000_0000.

Otherwise, the result is the value rnd, and an Inexact exception occurs if rnd is inexact (i.e., rnd is not equal to src).

The result is placed into doubleword element 0 of VSR[VRT+32] in signed integer format.

The contents of doubleword element 1 of VSR[VRT+32] are set to 0.

FPRF is set to undefined. FR is set to 0. FI is set to indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

See Table 76, “Actions for xscvqp swz,” on page 702.

Special Registers Altered:
  FPRF (undefined) FR (set to 0) FI
  FX VXSNAN VXCVI XX
VSR Data Layout for xscvqpswz

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32], dword[0]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned? (tr-unc(bool, src) / src)</td>
</tr>
<tr>
<td>FPSCR.VE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>0 -</th>
<th>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 -</td>
<td>fr(0), fi(0), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>0 -</td>
<td>T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(VXCVI)</td>
</tr>
<tr>
<td>-</td>
<td>fr(0), fi(1), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src = Nmin</td>
<td>- -</td>
<td>no T(Nmin), fr(0), fi(0), fprf(0bUUUUU)</td>
</tr>
<tr>
<td>src &gt; Nmin</td>
<td>- -</td>
<td>no T(Nmin), fr(0), fi(1), fprf(0bUUUUU)</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>- -</td>
<td>no T(Nmax), fr(0), fi(0), fprf(0bUUUUU)</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>0 -</td>
<td>T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(VXCVI)</td>
</tr>
<tr>
<td>-</td>
<td>fr(0), fi(1), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src = Nmax+1</td>
<td>0 -</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</td>
</tr>
<tr>
<td>1 -</td>
<td>fr(0), fi(0), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0 -</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</td>
</tr>
<tr>
<td>1 -</td>
<td>fr(0), fi(0), fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0 -</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)</td>
</tr>
<tr>
<td>1 -</td>
<td>fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
</tr>
</tbody>
</table>

Explanation:

\( T(x) \) Places the value \( x \) into the target VSR.
\( \text{VSR[VRT+32], dword[0]} + x \)
\( \text{VSR[VRT+32], dword[1]} + 0x0000_0000_0000_0000 \)

\( N\text{min} \) The smallest signed integer word value, \(-2^{31} (0x8000_8000_8000_8000)\).
\( N\text{max} \) The largest signed integer word value, \(2^{31} - 1 (0x7FFF_7FFF_7FFF_7FFF)\).
\( \text{src} \) The quad-precision floating-point value in VSR[VRB+32].

\( f2i(x) \) The quad-precision floating-point integer value \( x \) is converted to 32-bit signed integer format.
\( fx(x) \) FPSCR.FX is set to 1 if FPSCR.x=1. FPSCR.x is set to 1.
\( fi(x) \) FPSCR.FI is set to the value \( x \).
\( fr(x) \) FPSCR.FR is set to the value \( x \).
\( fprf(x) \) FPSCR.FPRF is set to the value \( x \).
\( \text{error()} \) The system error handler is invoked for the trap-enabled exception if MSR.FE0 or MSR.FE1 are set to any mode other than the ignore-exception mode.
\( \text{trunc}(x) \) Return the floating-point value \( x \) truncated to a floating-point integer.

Table 76. Actions for xscvqpswz
VSX Scalar Convert with round to zero
Quad-Precision to Unsigned Doubleword
format X-form

Let \( src \) be the quad-precision floating-point value in
\( VSR[VRT+32] \).

If \( src \) is a Signalling NaN, an Invalid Operation
exception occurs and \( VXSNAN \) and \( VXCVI \) are set to 1.

If \( src \) is a Quiet NaN or an Infinity, an Invalid Operation
exception occurs and \( VXCVI \) is set to 1.

If \( src \) is a NaN, the result is \( 0x0000_0000_0000_0000 \).

Otherwise, if \( src \) is a Zero, the result is
\( 0x0000_0000_0000_0000 \).

Otherwise, if \( src \) is a positive Infinity, the result is
\( 0xFFFF_FFFF_FFFF_FFFF \).

Otherwise, if \( src \) is a negative Infinity, the result is
\( 0x0000_0000_0000_0000 \).

Otherwise, do the following.

Let \( rnd \) be the value \( src \) truncated to a
floating-point integer.

If \( rnd \) is greater than \( +2^{64} - 1 \), an Invalid Operation
exception occurs, \( VXCVI \) is set to 1, and the result is
\( 0xFFFF_FFFF_FFFF_FFFF \).

Otherwise, if \( rnd \) is less than 0, an Invalid
Operation exception occurs, \( VXCVI \) is set to 1, and
the result is \( 0x0000_0000_0000_0000 \).

Otherwise, the result is the value \( rnd \), and an
Inexact exception occurs if \( rnd \) is inexact (i.e., \( rnd \)
is not equal to \( src \)).

The result is placed into doubleword element 0 of
\( VSR[VRT+32] \) in unsigned integer format.

The contents of doubleword element 1 of \( VSR[VRT+32] \)
are set to 0.

\( FPRF \) is set to undefined. \( FR \) is set to 0. \( FI \) is set to
indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, \( FR \) and \( FI \) are
set to 0.

If a trap-enabled Invalid Operation exception occurs,
\( VSR[VRT+32] \) and \( FPRF \) are not modified.

See Table 77, “Actions for xscvqpuzd,” on page 704.

Special Registers Altered:

\( FPRF \) (undefined) \( FR \) (set to 0) \( FI \)
\( VXSNAN \) \( VXCVI \) \( XX \)

<table>
<thead>
<tr>
<th>src</th>
<th>VRT</th>
<th>VRB</th>
<th>836</th>
</tr>
</thead>
<tbody>
<tr>
<td>061</td>
<td>87</td>
<td>17</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

\( src \leftarrow \) bfp_CONVERT_FROM_BFP128(VSR[VRT+32])

if src.class.QNaN=1 | src.class.SNaN=1 then do
\( result \leftarrow 0x0000_0000_0000_0000 \)
\( vsnann_flag \leftarrow src\text{.class}.SNaN \)
\( vcvi_flag \leftarrow 1 \)
end
else if src.class.Infinity=1 then do
\( vxcvi_flag \leftarrow 1 \)
if src.sign=0 then
\( result \leftarrow 0xFFFF_FFFF_FFFF_FFFF \)
else
\( result \leftarrow 0x0000_0000_0000_0000 \)
end
else if src.class.Zero then \( result \leftarrow 0x0000_0000_0000_0000 \)
else do
\( rnd \leftarrow \) bfp_ROUND_TO_INTEGER(0b001,src)
if bfp_COMPARE_GT(rnd, +2^{64}-1) then do
\( result \leftarrow 0xFFFF_FFFF_FFFF_FFFF \)
\( vxcvi_flag \leftarrow 1 \)
end
else if bfp_COMPARE_LT(rnd, 0) then do
\( result \leftarrow 0x0000_0000_0000_0000 \)
\( vxcvi_flag \leftarrow 1 \)
end
else do
\( result \leftarrow \) ui64_CONVERT_FROM_BFP(rnd)
if xx_flag=1 then SetFX(FPSCR.XX)
end
if vxsnann_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)

\( vx_flag \leftarrow \) vxsnann_flag | vxcvi_flag
\( ex_flag \leftarrow \) FPSCR.VE & vx_flag

if ex_flag=0 then do
\( VSR[VRT+32].dword[0] \leftarrow result \)
\( VSR[VRT+32].dword[1] \leftarrow 0x0000_0000_0000_0000 \)
FPSCR.FPRF \leftarrow 0bUUUUU
end
FPSCR.FR \leftarrow \{vx_flag=0 & inc_flag\}
FPSCR.FI \leftarrow \{vx_flag=0 & xx_flag\}
VSR Data Layout for xscvqpudz

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32]</td>
<td>VSR[VRT+32].dword[0]</td>
</tr>
</tbody>
</table>

### Returned Results and Status Setting

| src ≤ Nmin-1 | 0 | – | \( T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI) \) |
| Nmin-1 < src < Nmin | 1 | no | \( T(Nmin), fr(0), fi(0), fprf(0bUUUUU) \) |
| src = Nmin | – | no | \( T(Nmin), fr(0), fi(0), fprf(0bUUUUU) \) |
| Nmin < src < Nmax | 0 | yes | \( T(f2i(trunc(src))), fr(0), fi(0), fprf(0bUUUUU), fx(XX) \) |
| src = Nmax | – | no | \( T(Nmax), fr(0), fi(0), fprf(0bUUUUU) \) |
| Nmax < src < Nmax+1 | 0 | yes | \( T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI) \) |
| src ≥ Nmax+1 | 1 | yes | \( T(Nmax), fr(0), fi(0), fprf(0bUUUUU) \) |
| src is a QNaN | 0 | yes | \( T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI) \) |
| src is a SNaN | 1 | yes | \( T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN) \) |

<table>
<thead>
<tr>
<th>Explanation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(x) )</td>
</tr>
<tr>
<td>( VSR[VRT+32].dword[0] )</td>
</tr>
<tr>
<td>( Nmin )</td>
</tr>
<tr>
<td>( Nmax )</td>
</tr>
<tr>
<td>( src )</td>
</tr>
<tr>
<td>( f2i(x) )</td>
</tr>
<tr>
<td>( fx(x) )</td>
</tr>
<tr>
<td>( fr(x) )</td>
</tr>
<tr>
<td>( fprf(x) )</td>
</tr>
<tr>
<td>( error() )</td>
</tr>
<tr>
<td>( trunc(x) )</td>
</tr>
</tbody>
</table>

Table 77. Actions for xscvqpudz
VSX Scalar Convert with round to zero
Quad-Precision to Unsigned Quadword X-form

Let \( x \) be the quad-precision floating-point value in \( VSR[VRB+32] \).

If \( x \) is a Signalling NaN, an Invalid Operation exception occurs and \( VXSNAN \) and \( VXCVI \) are set to 1.

If \( x \) is a Quiet NaN or an Infinity, an Invalid Operation exception occurs and \( VXCVI \) is set to 1.

If \( x \) is a NaN, the result is 0.

Otherwise, if \( x \) is a Zero, the result is 0.
Otherwise, if \( x \) is \( +\)Infinity, the result is \( 2^{128} - 1 \).
Otherwise, if \( x \) is \( -\)Infinity, the result is 0.

Otherwise, do the following.

Let \( \text{src} \) be the quad-precision floating-point value in \( VSR[VRB+32] \).

\[
\begin{align*}
\text{src} & \leftarrow \text{bfp}_\text{CONVERT FROM BFP}_128(VSR[VRB+32]) \\
\text{if src.class.QNaN} = 1 \ | \ x_{\text{NaN}} = 1 \text{ then do} \\
& \quad \text{result} \leftarrow 0x0000_0000_0000_0000_0000_0000_0000_0000 \\
& \quad \text{vxsnan_flag} \leftarrow \text{src.class.SNaN} \\
& \quad \text{vxcvi_flag} \leftarrow 1
\end{align*}
\]

end
else if src.class.Infinity=1 then do
\[
\begin{align*}
& \quad \text{vxcvi_flag} \leftarrow 1 \\
& \quad \text{if src.sign=0 then} \\
& \quad \quad \text{result} \leftarrow 0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF \end{align*}
\]
else if src.class.Zero=1 then
\[
\begin{align*}
& \quad \text{result} \leftarrow 0x0000_0000_0000_0000_0000_0000_0000_0000 \\
& \quad \text{vxsnan_flag} \leftarrow \text{src.class.SNaN} \\
& \quad \text{vxcvi_flag} \leftarrow 1
\end{align*}
\]
end
else do
\[
\begin{align*}
& \quad \text{result} \leftarrow \text{bfp}_\text{ROUND TO INTEGER}(\text{bfp}_\text{ROUND_TO_INTEGER}(\text{bfp}_\text{ROUND_TO_INTEGER}(\text{bfp}_\text{ROUND TO INTEGER}(\text{bfp}_\text{ROUND TO INTEGER}(\text{bfp}_\text{ROUND TO INTEGER}(0b001, \text{src})))))) \\
& \quad \text{if bfp}_\text{COMPARE_GT}(\text{rnd}, +2^{128} - 1) \text{ then do} \\
& \quad \quad \text{result} \leftarrow 0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF \\
& \quad \quad \text{vxcvi_flag} \leftarrow 1
\end{align*}
\]
end
else if bfp_COMPARE_LT(rnd, 0) then do
\[
\begin{align*}
& \quad \text{result} \leftarrow 0x0000_0000_0000_0000_0000_0000_0000_0000 \\
& \quad \text{vxcvi_flag} \leftarrow 1
\end{align*}
\]
end
else do
\[
\begin{align*}
& \quad \text{result} \leftarrow \text{bfp}_\text{ROUND TO INTEGER}(\text{bfp}_\text{ROUND TO INTEGER}(\text{bfp}_\text{ROUND_TO_INTEGER}(\text{bfp}_\text{ROUND_TO_INTEGER}(\text{bfp}_\text{ROUND TO INTEGER}(\text{bfp}_\text{ROUND TO INTEGER}(0b001, \text{src})))))) \\
& \quad \text{if bfp}_\text{COMPARE_GT}(\text{rnd}, 0) \text{ then do} \\
& \quad \quad \text{result} \leftarrow 0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF \\
& \quad \quad \text{vxcvi_flag} \leftarrow 1
\end{align*}
\]
end
else do
\[
\begin{align*}
& \quad \text{result} \leftarrow \text{si128}_\text{CONVERT FROM BFP}(\text{rnd}) \\
& \quad \text{if xx_flag=1 then} \text{SetFX}(\text{FPSCR.XX})
\end{align*}
\]
end

if vxsnan_flag=1 then \text{SetFX}(\text{FPSCR.VXSNAN})
if vxcvi_flag=1 then \text{SetFX}(\text{FPSCR.VXCVI})
if \text{ex_flag}=0 then do
\[
\begin{align*}
& \quad \text{result} \leftarrow \text{VSR}[VRT+32] \leftarrow \text{result} \\
& \quad \text{FPSCR.FPRF} \leftarrow 0bUUUUU \\
& \quad \text{FPSCR.FR} \leftarrow (\text{vx_flag}=0) \& \text{inc_flag} \\
& \quad \text{FPSCR.FI} \leftarrow (\text{vx_flag}=0) \& \text{xx_flag}
\end{align*}
\]
end

\[
\begin{array}{cccccccccccc}
0 & 6 & 11 & 16 & 21 & 31 & 63 & VRT & 0 & VRB & 836 & /
\end{array}
\]

VSR Data Layout for xscvqpuz

Let \( src \) be the quad-precision floating-point value in \( VSR[VRB+32] \).

If \( src \) is a Signalling NaN, an Invalid Operation exception occurs and \( VXSNAN \) and \( VXCVI \) are set to 1.

If \( src \) is a Quiet NaN or an Infinity, an Invalid Operation exception occurs and \( VXCVI \) is set to 1.

If \( src \) is a NaN, the result is 0.

Otherwise, if \( src \) is a Zero, the result is 0.
Otherwise, if \( src \) is \( +\)Infinity, the result is \( 2^{128} - 1 \).
Otherwise, if \( src \) is \( -\)Infinity, the result is 0.

Otherwise, do the following.

Let \( \text{rnd} \) be the value \( src \) truncated to a floating-point integer.

If \( \text{rnd} \) is greater than \( 2^{128} - 1 \), an Invalid Operation exception occurs, \( VXCVI \) is set to 1, and the result is \( 2^{128} - 1 \).

Otherwise, if \( \text{rnd} \) is less than 0, an Invalid Operation exception occurs, \( VXCVI \) is set to 1, and the result is 0.

Otherwise, the result is the value \( \text{rnd} \), and an Inexact exception occurs if \( \text{rnd} \) is inexact (i.e., \( \text{rnd} \) is not equal to \( src \)).

The result is placed into \( VSR[VRT+32] \) in unsigned integer format.

\( \text{FPRF} \) is set to undefined. \( \text{FR} \) is set to 0. \( \text{FI} \) is set to indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, \( \text{FR} \) and \( \text{FI} \) are set to 0.

If a trap-enabled Invalid Operation exception occurs, \( VSR[VRT+32] \) and \( \text{FPRF} \) are not modified.

See Table 77, “Actions for xscvqpudz,” on page 704.

Special Registers Altered:
\[
\begin{align*}
\text{FPRF} & \quad \text{(undefined)} \\
\text{FR} & \quad \text{(set to 0)} \\
\text{FI} & \quad \text{( lessen if the rounded result is inexact)}
\end{align*}
\]

If an Invalid Operation exception occurs, \( \text{FR} \) and \( \text{FI} \) are set to 0.

If a trap-enabled Invalid Operation exception occurs, \( VSR[VRT+32] \) and \( \text{FPRF} \) are not modified.

See Table 77, “Actions for xscvqpudz,” on page 704.

Special Registers Altered:
\[
\begin{align*}
\text{FPRF} & \quad \text{(undefined)} \\
\text{FR} & \quad \text{(set to 0)} \\
\text{FI} & \quad \text{(lessen if the rounded result is inexact)}
\end{align*}
\]

If an Invalid Operation exception occurs, \( \text{FR} \) and \( \text{FI} \) are set to 0.

If a trap-enabled Invalid Operation exception occurs, \( VSR[VRT+32] \) and \( \text{FPRF} \) are not modified.
### FPSCR.E I

#### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src ≤ Nmin</th>
<th>FPSCRE</th>
<th>FPSCRAE</th>
<th>Error? (trunc(src) ≠ src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(0), fr(0), fx(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin-1</td>
<td>0</td>
<td>yes</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin-1</td>
<td>1</td>
<td>yes</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src ≥ Nmin</th>
<th>FPSCRE</th>
<th>FPSCRAE</th>
<th>Error? (trunc(src) ≠ src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src = Nmin</th>
<th>FPSCRE</th>
<th>FPSCRAE</th>
<th>Error? (trunc(src) ≠ src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src = Nmax</th>
<th>FPSCRE</th>
<th>FPSCRAE</th>
<th>Error? (trunc(src) ≠ src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src ≥ Nmax</th>
<th>FPSCRE</th>
<th>FPSCRAE</th>
<th>Error? (trunc(src) ≠ src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

#### Explanation:

- **T(x)** Places the value `x` into the target VSR.
- **Nmin** The smallest unsigned integer doubleword value, `0x0000_0000_0000_0000_0000_0000_0000_0000`.
- **Nmax** The largest unsigned integer doubleword value, `0x7FFF_7FFF_7FFF_7FFF_7FFF_7FFF_7FFF_7FFF`.
- **f2i(x)** The quad-precision floating-point integer value `x` is converted to 128-bit unsigned integer format.
- **fx** FPSCR.FX is set to 1 if FPSCR.xF = 0.
- **fi** FPSCR.FI is set to the value `x`.
- **fr** FPSCR.FR is set to the value `x`.
- **fprf** FPSCR.FPRF is set to the value `x`.
- **error()** The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
- **trunc(x)** Return the floating-point value `x` truncated to a floating-point integer.

### Table 78. Actions for `xscvqpuqz`
Let \( \text{src} \) be the quad-precision floating-point value in \( VSR[VRB+32] \).

If \( \text{src} \) is a Signalling NaN, an Invalid Operation exception occurs and \( VXSNAN \) and \( VXCVI \) are set to 1.

If \( \text{src} \) is a Quiet NaN or an Infinity, an Invalid Operation exception occurs and \( VXCVI \) is set to 1.

If \( \text{src} \) is a NaN, the result is \( 0x0000_0000_0000_0000 \).

Otherwise, if \( \text{src} \) is a Zero, the result is \( 0x0000_0000_0000_0000 \).

Otherwise, if \( \text{src} \) is a positive Infinity, the result is \( 0x0000_0000_FFFF_FFFF \).

Otherwise, do the following.

Let \( \text{rnd} \) be the value \( \text{src} \) truncated to a floating-point integer.

If \( \text{rnd} \) is greater than \( +2^{32}-1 \), an Invalid Operation exception occurs and \( VXCVI \) is set to 1, and the result is \( 0x0000_0000_FFFF_FFFF \).

Otherwise, if \( \text{rnd} \) is less than 0, an Invalid Operation exception occurs, \( VXCVI \) is set to 1, and the result is \( 0x0000_0000_0000_0000 \).

Otherwise, the result is the value \( \text{rnd} \), and an Inexact exception occurs if \( \text{rnd} \) is inexact (i.e., \( \text{rnd} \) is not equal to \( \text{src} \)).

The result is placed into doubleword element 0 of \( VSR[VRT+32] \) in unsigned integer format.

The contents of doubleword element 1 of \( VSR[VRT+32] \) are set to 0.

\( FPRF \) is set to undefined. \( FR \) is set to 0. \( FI \) is set to indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, \( FR \) and \( FI \) are set to 0.

If a trap-enabled Invalid Operation exception occurs, \( VSR[VRT+32] \) and \( FPRF \) are not modified.


**Special Registers Altered:**

\[ FPRF \text{ (undefined)} \quad FR \text{ (set to 0)} \quad FI \]

\[ FX VXSNAN VXCVI XX \]
### VSR Data Layout for xscvqpuwz

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
<th>tgt</th>
<th>VSR[VRT+32].dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
</table>

### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>FPSCR VE</th>
<th>FPSCR XE</th>
<th>X</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤ Nmin-1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>0</td>
<td>yes</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(xx), error()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(xx), error()</td>
<td></td>
</tr>
<tr>
<td>≥ Nmin</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fr(0), fi(0), fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

#### Explanation:

- **T(x)**: Places the value x into the target VSR.
  - `VSR[VRT+32], dword[0] = x`
  - `VSR[VRT+32], dword[1] = 0x0000_0000_0000_0000`

- **Nmin**: The smallest unsigned integer word value, 0 (0x0000_0000_0000_0000).
- **Nmax**: The largest unsigned integer word value, 2\(^{32}\) - 1 (0x0000_0000_FFFF_FFFF).
- **src**: The quad-precision floating-point value in VSR[VRB+32].
- **f2i(x)**: The quad-precision floating-point integer value x is converted to 32-bit unsigned integer format.
- **fx(x)**: FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
- **fr(x)**: FPSCR.FR is set to the value x.
- **fprf(x)**: FPSCR.FPRF is set to the value x.
- **error()**: The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
- **trunc(x)**: Return the floating-point value x truncated to a floating-point integer.

---

Table 79. Actions for xscvqpuwz
### VSX Scalar Convert Single-Precision to Double-Precision format XX2-form

**xscvspdp XT,XB**

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>11</th>
<th>B</th>
<th>329</th>
<th>90</th>
</tr>
</thead>
</table>

If MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

\[
\text{src} \leftarrow \text{bfp_CONVERT_FROM_BFP32(VSR[32\times BX+B].word[0])}
\]

\[
\text{vxsnan_flag} \leftarrow \text{src.class.SNaN}
\]

\[
\text{result} \leftarrow \text{bfp64_CONVERT_FROM_BFP(src)}
\]

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

\[
\text{vex_flag} \leftarrow \text{FPSCR.VE} \& \text{vxsnan_flag}
\]

\[
\text{FPSCR.FR} \leftarrow 0b0
\]

\[
\text{FPSCR.FI} \leftarrow 0b0
\]

if vex_flag=0 then do

\[
\text{VSR[32\times TX+T].dword[0]} \leftarrow \text{result}
\]

\[
\text{VSR[32\times TX+T].dword[1]} \leftarrow 0x0000_0000_0000_0000
\]

\[
\text{FPSCR.FPRF} \leftarrow \text{fprf_CLASS_BFP64(result)}
\]

end

Let XT be the value 32×TX + T.

Let XB be the value 32×BX + B.

Let src be the single-precision floating-point value in word element 0 of VSR[XB].

If src is a SNaN, the result is src, converted to a QNaN (i.e., bit 9 of src set to 1). VXSNAN is set to 1.

Otherwise, the result is src.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to 0, FI is set to 0.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified, FPRF is not modified, FR is set to 0, and FI is set to 0.

**Special Registers Altered**

FPRF FR=0b0 FI=0b0 FX VXSNAN

---

### VSR Data Layout for xscvspdp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].word[0]</th>
<th>unused</th>
<th>unused</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XT].dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
</table>

**Programming Note**

**xscvspdp** can be used to convert a single-precision value in single-precision format to double-precision format for use by Floating-Point scalar single-precision operations.

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form

xscvspdpn  XT,XB

```
if MSR.VSX=0 then VSX_Unavailable();
reset_xflags();

src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[0])
result ← bfp64_CONVERT_FROM_BFP(src)
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the single-precision floating-point value in word element 0 of VSR[XB].

src is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered
None

--- Programming Note ---

**xscvspdp** should be used to convert a vector single-precision floating-point value to scalar double-precision format.

**xscvspdpn** should be used to convert a vector single-precision floating-point value to scalar single-precision format.

--- Programming Note ---

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xscvspdpn

```
src  VSR[XB].word[0] unused unused
tgt  VSR[XT].dword[0] 0x0000_0000_0000_0000
```

### VSX Scalar Convert with round Signed Quadword to Quad-Precision X-form

**xscvsqqp**

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>62</th>
<th>61</th>
<th>60</th>
<th>59</th>
<th>58</th>
<th>57</th>
<th>56</th>
<th>55</th>
<th>54</th>
<th>53</th>
<th>52</th>
<th>51</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRB</td>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>836</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

\[
\text{src} \leftarrow \text{bfp\_CONVERT\_FROM\_SI128(VSR[VRB+32])}
\]

\[
\text{rnd} \leftarrow \text{bfp\_ROUND\_TO\_BFP128(0, FPSCR.RN, src)}
\]

\[
\text{result} \leftarrow \text{bfp128\_CONVERT\_FROM\_BFP(rnd)}
\]

If xx_flag=1 then SetFX(XX)

\[
\text{VSR}[VRT+32] \leftarrow \text{result}
\]

\[
\text{FPSCR.FPRF} \leftarrow \text{fprf\_CLASS\_BFP128(result)}
\]

\[
\text{FPSCR.FR} \leftarrow \text{inc\_flag}
\]

\[
\text{FPSCR.FI} \leftarrow \text{xx\_flag}
\]

Let \( \text{src} \) be the 128-bit signed integer value in \( \text{VSR}[\text{VRB}+32] \).

\( \text{src} \) is converted to an unbounded-precision floating-point value and rounded to quad-precision using the rounding mode specified by \( \text{RN} \).

The result is placed into \( \text{VSR}[\text{VRT}+32] \) in quad-precision format.

\( \text{FPRF} \) is set to the class and sign of the result. \( \text{FR} \) is set to indicate if the result was incremented when rounded. \( \text{FI} \) is set to indicate the result is inexact.

**Special Registers Altered:**

\[ \text{FPRF FR FI FX XX} \]

### VSR Data Layout for xscvsqqp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>0</td>
<td>127</td>
</tr>
</tbody>
</table>
**VSX Scalar Convert with round Signed Doubleword to Double-Precision format XX2-form**

<table>
<thead>
<tr>
<th>xscvsxddp</th>
<th>XT,XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>60</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_SI64(VSR[32×BX+B].dword[0])

rnd ← bfp_ROUND_TO_BFP64(0b0, FPSCR.RN, v)

result ← bfp64_CONVERT_FROM_BFP(rnd)

if xx_flag=1 then SetFX(FPSCR.XX)

VSR[32×TX+T].dword[0] ← result

VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000

FPSCR.FPRF ← fprf_CLASS_BFP64(result)

FPSCR.FR ← inc_flag

FPSCR.FI ← xx_flag

Let XT be the value 32×TX + T.

Let XB be the value 32×BX + B.

Let src be the signed integer value in doubleword element 0 of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

**Special Registers Altered**

FPRF FR FI FX XX

**VSR Data Layout for xscvsxddp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

---

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
VSX Scalar Convert with round Signed Doubleword to Single-Precision format

**XX2-form**

```plaintext
xscvsxdsp XT, XB
```

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>//</th>
<th>B</th>
<th>21</th>
<th>B[31]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>312</td>
<td>312</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

src ← bfp_CONVERT_FROM_SI64(VSR[XB].dword[0])
rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if xx_flag=1 then SetFX(FPSCR.XX)

VSR[32×TX+T].dword[0] ← result64
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
FPSCR.FR ← inc_flag
FPSCR.FI ← xx_flag

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the two's-complement integer value in doubleword element 0 of VSR[XB].

csrc is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPFRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

**Special Registers Altered**

FPFRF FR FI FX XX

**VSR Data Layout for xscvsxdsp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XT].dword[0]</th>
<th>6x0000_0000_0000_0000</th>
</tr>
</thead>
</table>

---

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

---

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the two's-complement integer value in doubleword element 0 of VSR[XB].

csrc is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPFRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

**Special Registers Altered**

FPFRF FR FI FX XX

**VSR Data Layout for xscvsxdsp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XT].dword[0]</th>
<th>6x0000_0000_0000_0000</th>
</tr>
</thead>
</table>
VSX Scalar Convert Signed Doubleword to Quad-Precision format X-form

\( \text{xscvsdqp} \quad \text{VRT}, \text{VRB} \)

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>10</th>
<th>VRB</th>
<th>836</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

\( \text{src} \leftarrow \text{bfp}_\text{CONV} \text{ERT}, \text{FROM}_\text{SI64}(\text{VSR}[\text{VRB+32}]. \text{dword}[0]) \)

\( \text{result} \leftarrow \text{bfp128}_\text{CONV} \text{ERT}, \text{FROM}_\text{BFP}(\text{src}) \)

\( \text{VSR}[\text{VRT+32}] \leftarrow \text{result} \)

\( \text{FPSCR.FPRF} \leftarrow \text{fprf}_\text{CLASS}_\text{BFP128}(\text{result}) \)

\( \text{FPSCR.FR} \leftarrow 0 \)

\( \text{FPSCR.FI} \leftarrow 0 \)

Let \( \text{src} \) be the signed integer value in doubleword element 0 of \( \text{VSR}[\text{VRB+32}] \).

\( \text{src} \) is placed into \( \text{VSR}[\text{VRT+32}] \) in quad-precision floating-point format.

\( \text{FPREF} \) is set to the class and sign of the result. \( \text{FR} \) is set to 0. \( \text{FI} \) is set to 0.

Special Registers Altered:

\( \text{FPREF FR (set to 0) FI (set to 0)} \)

VSR Data Layout for xscvsdqp & xscvudqp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>

VSX Scalar Convert Unsigned Doubleword to Quad-Precision format X-form

\( \text{xscvudqp} \quad \text{VRT}, \text{VRB} \)

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>2</th>
<th>VRB</th>
<th>836</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

\( \text{src} \leftarrow \text{bfp}_\text{CONV} \text{ERT}, \text{FROM}_\text{UI64}(\text{VSR}[\text{VRB+32}]. \text{dword}[0]) \)

\( \text{result} \leftarrow \text{bfp128}_\text{CONV} \text{ERT}, \text{FROM}_\text{BFP}(\text{src}) \)

\( \text{VSR}[\text{VRT+32}] \leftarrow \text{result} \)

\( \text{FPSCR.FPRF} \leftarrow \text{fprf}_\text{CLASS}_\text{BFP128}(\text{result}) \)

\( \text{FPSCR.FR} \leftarrow 0 \)

\( \text{FPSCR.FI} \leftarrow 0 \)

Let \( \text{src} \) be the unsigned integer value in doubleword element 0 of \( \text{VSR}[\text{VRB+32}] \).

\( \text{src} \) is placed into \( \text{VSR}[\text{VRT+32}] \) in quad-precision floating-point format.

\( \text{FPREF} \) is set to the class and sign of the result. \( \text{FR} \) is set to 0. \( \text{FI} \) is set to 0.

Special Registers Altered:

\( \text{FPREF FR (set to 0) FI (set to 0)} \)
VSX Scalar Convert with round Unsigned Quadword to Quad-Precision format X-form

```c
xscvuqqp  VRT,VRB

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

src ← bfp_CONVERT_FROM_UI128(VSR[VRT+32])
rnd ← bfp_ROUND_TO_BFP256(0b0,FPSCR.RN,src)
result ← bfp256_CONVERT_FROM_BFP(rnd)
if xx_flag=1 then SetFX(XX)
VSR[VRT+32] ← result
FPSCR.FPRF ← fprf_CLASS_BFP256(result)
FPSCR.FR ← inc_flag
FPSCR.FI ← xx_flag
```

Let `src` be the 128-bit unsigned integer value in VSR[VRT+32].

`src` is converted to an unbounded-precision floating-point value and rounded to quad-precision using the rounding mode specified by RN.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

**Special Registers Altered:**

- FPRF
- FR
- FI
- FX
- XX

---

VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format XX2-form

```c
xscvuxddp  XT,XB

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

src ← bfp_CONVERT_FROM_UI64(VSR[32×TX+T].dword[0])
rnd ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)
result ← bfp64_CONVERT_FROM_BFP(rnd)
if xx_flag=1 then SetFX(FPSCR.XX)
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP64(result)
FPSCR.FR ← inc_flag
FPSCR.FI ← xx_flag
```

Let `XT` be the value 32×TX + T.
Let `XB` be the value 32×BX + B.

Let `src` be the unsigned integer value in doubleword element 0 of VSR[XB].

`src` is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

**Special Registers Altered**

- FPRF
- FR
- FI
- FX
- XX

---

### VSR Data Layout for xscvuqqp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

### VSR Data Layout for xscvuxddp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRT+32].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>
VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format

**XX2-form**

```plaintext
xscvuxdsp XT,XB
```

If MSR.VSX=0 then VSX_Unavailable()

```plaintext
reset_xflags()
```

```plaintext
src ← bfp_CONVERT_FROM_UI64(VSR[32×BX+B].dword[0])
```

```plaintext
rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)
```

If xx_flag=1 then SetFX(FPSCR.XX)

```plaintext
VSR[32×TX+T].dword[0] ← result64
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
FPSCR.FR ← inc_flag
FPSCR.FI ← xx_flag
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the unsigned-integer value in doubleword element 0 of VSR[XB].

src is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPFR is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

**Special Registers Altered**

FPFR FR FI FX XX

**VSR Data Layout for xscvuxdsp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

---

Programming Note

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the unsigned-integer value in doubleword element 0 of VSR[XB].

src is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPFR is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

**Special Registers Altered**

FPFR FR FI FX XX

**VSR Data Layout for xscvuxdsp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>
### VSX Scalar Divide Double-Precision XX3-form

**xsdivdp**  
**XT,XA,XB**

1. If MSR.VSX=0 then VSX_Unavailable()
2. reset_xflags()

```
src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v ← bfp_DIVIDE(src1,src2)
rnd ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)
result ← bfp64_CONVERT_FROM_BFP(rnd)
```

3. If vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
4. If vxidi_flag=1 then SetFX(FPSCR.VXIDI)
5. If vxzdz_flag=1 then SetFX(FPSCR.VXZDZ)
6. If ox_flag=1 then SetFX(FPSCR.OX)
7. If ux_flag=1 then SetFX(FPSCR.UX)
8. If xx_flag=1 then SetFX(FPSCR.XX)
9. If zx_flag=1 then SetFX(FPSCR.ZX)
10. vx_flag ← vxsnan_flag | vxidi_flag | vxzdz_flag
11. vex_flag ← FPSCR.VE & vx_flag
12. zex_flag ← FPSCR.ZE & zx_flag
13. If vex_flag=0 & zex_flag=0 then do
    VSR[32×TX+T].dword[0] ← result
    VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
    FPSCR.FR ← inc_flag
    FPSCR.FI ← xx_flag
14. Else do
    FPSCR.FR ← 0b1
    FPSCR.FI ← 0b1
15. end

Let **XT** be the value 32×TX + T.
Let **XA** be the value 32×AX + A.
Let **XB** be the value 32×BX + B.

#### Special Registers Altered

- FPRF
- FR
- FI
- FX
- OX
- UX
- XX
- VXSNAN
- VXIDI
- VXZDZ

---

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
## Table 80. Actions for xsdivdp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← vxxnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Zero</td>
<td>v ← D(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← D(src1,src2)</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Zero</td>
<td>v ← D(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← vxxnan_flag ← 1</td>
</tr>
<tr>
<td>NaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← vxxnan_flag ← 1</td>
</tr>
</tbody>
</table>

**Explanation:**
src1 The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2 The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN Default quiet NaN (0x7FF8_0000_0000_0000).
NZF Nonzero finite number.
D(x,y) Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.
Q(x) Return a QNaN with the payload of x.
v The intermediate result having unbounded significand precision and unbounded exponent range.
**VSX Scalar Divide Quad-Precision [using round to Odd] X-form**

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 and src2 are Infinity values, an Invalid Operation exception occurs and VXI DI is set to 1.

If src1 and src2 are Zero values, an Invalid Operation exception occurs and VXZDZ is set to 1.

If src1 is a finite value and src2 is a Zero value, an Zero Divide exception occurs and VXZ is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src1 is a Quiet NaN, the result is src2.

Otherwise, if src1 and src2 are Infinity values, or if src1 and src2 are Zero values, the result is the default Quiet NaN[1].

Otherwise, if src1 is a non-zero value and src2 is a Zero value, the result is an Infinity.

Otherwise, do the following.

The normalized quotient of src1 divided by src2 is produced with unbounded significand precision and exponent range.

See Table 81, “Actions for xsdivqp[0],” on page 720.

If the intermediate result is Tiny (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right N bits, where N is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.


The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-disabled Zero Divide exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception or a trap-enabled Zero Divide exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

---

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.
**Special Registers Altered:**

- PRF
- FX
- VXSNAN
- VXIDI
- VXZDZ
- OX
- UX
- ZX
- XX

**VSR Data Layout for xsdivqp[o]**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[ VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[ VRB+32]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[ VRT+32]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>x ← Infinity</td>
<td>v ← Infinity</td>
<td>v ← Infinity</td>
<td>x ← Infinity</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← De(sign1, src2)</td>
<td>v ← De(sign1, src2)</td>
<td>v ← De(sign1, src2)</td>
<td>v ← De(sign1, src2)</td>
<td>v ← De(sign1, src2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← +Zero</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td>v ← qNaN, vxzdz_flag ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1**: The quad-precision floating-point value in VSR[ VRA+32].
- **src2**: The quad-precision floating-point value in VSR[ VRB+32].
- **qNaN**: Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **De(x, y)**: The floating-point value x is divided by floating-point value y. Return the normalized quotient, having unbounded range and precision.
- **quiet(x)**: Convert x to the corresponding Quiet NaN.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.

**Table 81. Actions for xsdivqp[o]**

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then subtracted or added as appropriate, depending on the signs of the operands, to form an intermediate difference. All 64 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
VSX Scalar Divide Single-Precision XX3-form

xsdivsp \( XT,XA, XB \)

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>24</th>
<th>SB</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>25</td>
<td>30</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v ← bfp_DIVIDE(src1, src2)
rd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vzidi_flag=1 then SetFX(FPSCR.VXIDI)
if vxzdz_flag=1 then SetFX(FPSCR.VXZDZ)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)
if zx_flag=1 then SetFX(FPSCR.ZX)
vx_flag ← vsnan_flag | vzidi_flag | vxzdz_flag
vex_flag ← FPSCR.VE & vx_flag
zex_flag ← FPSCR.ZE & zx_flag

if vx_flag=1 & zex_flag=0 then do
VSR[32×TX+T].dword[0] ← result64
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
FPSCR.FR ← inc_flag
FPSCR.FI ← xx_flag
end
else do
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
end

Let \( XT \) be the value \( 32×TX + T \).
Let \( XA \) be the value \( 32×AX + A \).
Let \( XB \) be the value \( 32×BX + B \).

Let \( src1 \) be the double-precision floating-point value in doubleword element 0 of VSR[\( XA \)].

\( src1 \) is divided[1] by \( src2 \), producing a quotient having unbounded range and precision.

The quotient is normalized[2].

See Table 82, “Actions for xsdivsp,” on page 722.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.


The result is placed into doubleword element 0 of VSR[\( XT \)] in double-precision format.

The contents of doubleword element 1 of VSR[\( XT \)] are set to 0.

FPFRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[\( XT \)] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

Special Registers Altered

FPRF FR FI
FX OX UX ZX XX VXSNAN VXIDI VXZDZ

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Explanation:

**src1**  The double-precision floating-point value in doubleword element 0 of VSR[XA].

**src2**  The double-precision floating-point value in doubleword element 0 of VSR[XB].

dQNaN  Default quiet NaN (0x7FF8_0000_0000_0000).

NZF  Nonzero finite number.

D(x,y)  Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.

Q(x)  Return a QNaN with the payload of x.

v  The intermediate result having unbounded significand precision and unbounded exponent range.

---

### Table 82. Actions for xsdivsp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Zero</td>
<td>v ← D(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← D(src1,src2)</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Zero</td>
<td>v ← D(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← D(src1,src2)</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← vxsnan_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← vxsnan_flag ← 1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

This table outlines the actions taken during the execution of the xsdivsp instruction, detailing how the intermediate and final results are determined based on the input values and the conditions specified.
VSX Scalar Insert Exponent Double-Precision X-form

xsiexpdp XT,RA,RB

if MSR.VSX=0 then VSX_Unavailable()

src1 ← GPR[RA]
src2 ← GPR[RB]

VSR[32×TX+T].dword[0].bit[0] ← src1.bit[0]
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000

Let XT be the sum 32×TX + T.
Let src1 be the unsigned integer value in GPR[RA].
Let src2 be the unsigned integer value in GPR[RB].

The contents of bit 0 of src1 are placed into bit 0 of VSR[XT].

The contents of bits 53:63 of src2 are placed into bits 1:11 of VSR[XT].

The contents of bits 12:63 of src1 are placed into bits 12:63 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered:
None

Programming Note
This instruction can be used to produce a single-precision result.

Programming Note
Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
VSX Scalar Insert Exponent Quad-Precision X-form

\textit{xsiexpqp} VRT, VRA, VRB

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
 & 63 & VRT & VRA & 868 \\
\hline
0 & 6 & 11 & 16 & 21 & 31 \\
\hline
\end{tabular}
\end{center}

if MSR.VSX=0 then VSX_Unavailable()

\begin{verbatim}
VSRR[VRT+32].bit[0] ← VSRR[VRA+32].bit[0]
\end{verbatim}

The contents of bit 0 of \textit{VSR}[VRA+32] are placed into bit 0 of \textit{VSR}[VRT+32].

The contents of bit 49:63 of doubleword element 0 of \textit{VSR}[VRB+32] are placed into bits 1:15 of \textit{VSR}[VRT+32].

The contents of bit 16:127 of \textit{VSR}[VRA+32] are placed into bits 16:127 of \textit{VSR}[VRT+32].

Special Registers Altered:
None

VSR Data Layout for xsiexpqp

\begin{verbatim}
src1
\end{verbatim}

\begin{verbatim}
src2
\end{verbatim}

\begin{verbatim}
tgt
\end{verbatim}
Let $X_A$ be the value $32 \times X + A$.
Let $X_B$ be the value $32 \times X + B$.

Let $src_1$ be the double-precision floating-point value in doubleword element 0 of $VSR[X_A]$.

For $xsmaddadp$, do the following.
- Let $src_2$ be the double-precision floating-point value in doubleword element 0 of $VSR[X_T]$.
- Let $src_3$ be the double-precision floating-point value in doubleword element 0 of $VSR[X_B]$.

For $xsmaddmdp$, do the following.
- Let $src_2$ be the double-precision floating-point value in doubleword element 0 of $VSR[X_B]$.
- Let $src_3$ be the double-precision floating-point value in doubleword element 0 of $VSR[X_T]$.

$src_1$ is multiplied\(^1\) by $src_3$, producing a product having unbounded range and precision.

See part 1 of Table 83.

$src_2$ is added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 83.

The intermediate result is rounded to double-precision using the rounding mode specified by $RN$.


The result is placed into doubleword element 0 of $VSR[X_T]$ in double-precision format.

The contents of doubleword element 1 of $VSR[X_T]$ are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, $VSR[X_T]$ and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

**Special Registers Altered**

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
</table>
| 60 T A B 33 AX BX TX | 0 6 11 16 21 29 30 31 | 60 T A B 41 AX BX TX | 0 6 11 16 21 29 30 31

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

### VSR Data Layout for xsmaddadp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XT].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

### VSR Data Layout for xsmaddmdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XT].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>
**Table 83. Actions for xsmadd(a|m)dp**

<table>
<thead>
<tr>
<th>src3</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>src1 ⊕ +Infinity</td>
<td>src2 ⊕ +Infinity</td>
<td>src1 ⊕ +Zero</td>
<td>src2 ⊕ +Zero</td>
<td>src1 ⊕ +Null</td>
<td>src2 ⊕ +Null</td>
<td>src1 ⊕ +Null</td>
<td>src2 ⊕ +Null</td>
</tr>
<tr>
<td>-NZF</td>
<td>src1 ⊕ +Null</td>
<td>src2 ⊕ -Null</td>
<td>src1 ⊕ -Zero</td>
<td>src2 ⊕ -Zero</td>
<td>src1 ⊕ +Null</td>
<td>src2 ⊕ -Null</td>
<td>src1 ⊕ -Zero</td>
<td>src2 ⊕ +Zero</td>
</tr>
<tr>
<td>-Zero</td>
<td>src1 ⊕ +Infinity</td>
<td>src2 ⊕ +Infinity</td>
<td>src1 ⊕ -Infinity</td>
<td>src2 ⊕ -Infinity</td>
<td>src1 ⊕ +Infinity</td>
<td>src2 ⊕ +Infinity</td>
<td>src1 ⊕ +Infinity</td>
<td>src2 ⊕ +Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>src1 ⊕ -Zero</td>
<td>src2 ⊕ -Zero</td>
<td>src1 ⊕ +Infinity</td>
<td>src2 ⊕ +Infinity</td>
<td>src1 ⊕ -Zero</td>
<td>src2 ⊕ -Zero</td>
<td>src1 ⊕ +Infinity</td>
<td>src2 ⊕ +Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>src1 ⊕ +Null</td>
<td>src2 ⊕ +Null</td>
<td>src1 ⊕ +Null</td>
<td>src2 ⊕ +Null</td>
<td>src1 ⊕ -Infinity</td>
<td>src2 ⊕ -Infinity</td>
<td>src1 ⊕ +NZF</td>
<td>src2 ⊕ +NZF</td>
</tr>
<tr>
<td>+Infinity</td>
<td>src1 ⊕ -Infinity</td>
<td>src2 ⊕ -Infinity</td>
<td>src1 ⊕ +Infinity</td>
<td>src2 ⊕ +Infinity</td>
<td>src1 ⊕ -Infinity</td>
<td>src2 ⊕ -Infinity</td>
<td>src1 ⊕ +Infinity</td>
<td>src2 ⊕ +Infinity</td>
</tr>
</tbody>
</table>

**Explanation:**
- src1: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- src2: For xsmaddadp, the double-precision floating-point value in doubleword element 0 of VSR[XT].
- src3: For xsmadda, the double-precision floating-point value in doubleword element 0 of VSR[XB].
- dQNaN: Default quiet NaN (0x7FF8_0000_0000_0000).
- NZF: Nonzero finite number.
- Rezd: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- Q(x): Return a QNaN with the payload of x.
- A(x,y): Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- M(x,y): Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- p: The intermediate product having unbounded range and precision.
- v: The intermediate result having unbounded range and precision.
VSX Scalar Multiply-Add Type-A
Single-Precision XX3-form

\[
x_{\text{smaddasp}} \quad X_T, X_A, X_B
\]

Let \(X_T\) be the value \(32 \times X_T + T\).
Let \(X_A\) be the value \(32 \times X_A + A\).
Let \(X_B\) be the value \(32 \times X_B + B\).

For \(x_{\text{smaddasp}}\), do the following.

– Let \(src1\) be the double-precision floating-point value in doubleword element 0 of \(VSR[X_A]\).
– Let \(src2\) be the double-precision floating-point value in doubleword element 0 of \(VSR[X_T]\).
– Let \(src3\) be the double-precision floating-point value in doubleword element 0 of \(VSR[X_B]\).

For \(x_{\text{smaddmsp}}\), do the following.

– Let \(src1\) be the double-precision floating-point value in doubleword element 0 of \(VSR[X_A]\).
– Let \(src2\) be the double-precision floating-point value in doubleword element 0 of \(VSR[X_B]\).
– Let \(src3\) be the double-precision floating-point value in doubleword element 0 of \(VSR[X_T]\).

\(src1\) is multiplied\(^{[1]}\) by \(src3\), producing a product having unbounded range and precision.

See part 1 of Table 84, “Actions for \(x_{\text{smadd}(a|m)sp}\),” on page 730.

\(src2\) is added\(^{[2]}\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^{[3]}\).

See part 2 of Table 84, “Actions for \(x_{\text{smadd}(a|m)sp}\),” on page 730.

The intermediate result is rounded to single-precision using the rounding mode specified by \(RN\).


The result is placed into doubleword element 0 of \(VSR[X_T]\) in double-precision format.

The contents of doubleword element 1 of \(VSR[X_T]\) are set to 0.

\(FPFR\) is set to the class and sign of the result as represented in single-precision format. \(FR\) is set to indicate if the result was incremented when rounded. \(FI\) is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, \(VSR[X_T]\) and \(FPFR\) are not modified, and \(FR\) and \(FI\) are set to 0.

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

**Special Registers Altered**

<table>
<thead>
<tr>
<th>FRF</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>UX</th>
<th>XX</th>
<th>VXSNAN</th>
<th>VXISI</th>
<th>VXIMZ</th>
</tr>
</thead>
</table>

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

---

**VSR Data Layout for xsmaddasp**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td>VSR[XA].dword[0]</td>
<td></td>
</tr>
<tr>
<td>src2</td>
<td>VSR[XT].dword[0]</td>
<td></td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XB].dword[0]</td>
<td></td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

---

**VSR Data Layout for xsmaddmsp**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td>VSR[XA].dword[0]</td>
<td></td>
</tr>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td></td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XT].dword[0]</td>
<td></td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>
**Table 84. Actions for xsmadd(a|m)sp**

**Part 1: Multiply**

<table>
<thead>
<tr>
<th>src1</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← –Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>–NZF</td>
<td>p ← +Infinity</td>
<td>p ← –Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>–Zero</td>
<td>p ← +Infinity</td>
<td>p ← –Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← +Infinity</td>
<td>p ← –Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← +Infinity</td>
<td>p ← –Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← +Infinity</td>
<td>p ← –Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
</tr>
</tbody>
</table>

**Part 2: Add**

<table>
<thead>
<tr>
<th>src1</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>–NZF</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>–Zero</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>QNaN &amp; src1 is NaN</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
<td>v ← +p</td>
</tr>
</tbody>
</table>

**Explanation:**

- src1: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- src2: For xsmaddasp, the double-precision floating-point value in doubleword element 0 of VSR[XT].
  For xsmaddmsp, the double-precision floating-point value in doubleword element 0 of VSR[XB].
- src3: For xsmaddasp, the double-precision floating-point value in doubleword element 0 of VSR[XB].
- dQNaN: Default quiet NaN (0x7FF8_0000_0000_0000).
- NZF: Nonzero finite number.
- Rezd: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- Q(x): Return a QNaN with the payload of x.
- A(x,y): Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
  Note: If x = –y, v is considered to be an exact-zero-difference result (Rezd).
- M(x,y): Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- p: The intermediate product having unbounded range and precision.
- v: The intermediate result having unbounded range and precision.
VSX Scalar Multiply-Add Quad-Precision [using round to Odd] X-form

\[
\begin{align*}
&\text{xsmadd}p & & \text{VRT}, \text{VRA}, \text{VRB} \\
&\text{xsmadd}p & & \text{VRT}, \text{VRA}, \text{VRB}
\end{align*}
\]

\((R0=0)\)

\((R0=1)\)

\[
\begin{array}{cccccc}
63 & 6 & 11 & 16 & 21 & 388 \\
0 & 6 & 11 & 16 & 21 & 31
\end{array}
\]

if MSR.VSX=0 then VSX_Unavailable() reset_xflags() src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32]) src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRT+32]) src3 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32]) v ← bfp_MULTIPLY_ADD(src1, src3, src2) rnd ← bfp_ROUND_TO_BFP128(RD, FPSCR.RN, v) result ← bfp128_CONVERT_FROM_BFP(rnd) if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN) if vximz_flag=1 then SetFX(FPSCR.VXIMZ) if vxisi_flag=1 then SetFX(FPSCR.VXISI) if ox_flag=1 then SetFX(FPSCR.OX) if ux_flag=1 then SetFX(FPSCR.UX) if xx_flag=1 then SetFX(FPSCR.XX) vx_flag ← vxsnan_flag | vximz_flag | vxisi_flag ex_flag ← FPSCR.VE & vx_flag if ex_flag=0 then do VSR[VRT+32] ← result FPSCR.FPREF ← fprf_CLASS_BFP128(result) end FPSCR.FR ← (vx_flag=0) & inc_flag FPSCR.FI ← (vx_flag=0) & xx_flag

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRT+32] represented in quad-precision format.

Let src3 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1, src2, or src3 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If src2 and the product of src1 and src3 are Infinity values having opposite signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src3 is a Signalling NaN, the result is the Quiet NaN corresponding to src3.

Otherwise, if src3 is a Quiet NaN, the result is src3.

Otherwise, if src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, the result is the default Quiet NaN[1].

Otherwise, if the product of src1 and src3, and src2 are Infinity values having opposite signs, the result is the default Quiet NaN.

Otherwise, do the following. src1 is multiplied by src3, producing a product having unbonded significand precision and exponent range.

See part 1 of Table 83. "Actions for xsmadd(a|m)dp".

src2 is added to the product, producing a sum having unbonded range and precision.

See part 2 of Table 83. "Actions for xsmadd(a|m)dp".

If the intermediate result is Tiny (i.e., the unbiased exponent is less than \(-16382\)) and UE=0, the significand is shifted right \(N\) bits, where \(N\) is the difference between \(-16382\) and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value \(-16382\). If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.


The result is placed into VSR[VRT+32] in quad-precision format.

FPREF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

---

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000_0000.
If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPREF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

**Special Registers Altered:**

<table>
<thead>
<tr>
<th>PPREF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX</td>
<td>VXSNAN</td>
<td>VXIMZ</td>
</tr>
</tbody>
</table>

**VSR Data Layout for xsmaddqp[o]**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

0 127
### Part 1: Multiply

<table>
<thead>
<tr>
<th>src3</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>p ← –Infinity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td></td>
<td>p ← mul(src1, src3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–Zero</td>
<td>p ← –Zero</td>
<td>p ← +Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td></td>
<td></td>
<td>p ← +Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td></td>
<td></td>
<td></td>
<td>p ← –Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← –Infinity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← quiet(src3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**vximz_flag** ← 1

### Part 2: Add

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>v ← –Infinity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td></td>
<td>v ← add(p, src2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–Zero</td>
<td>v ← –Zero</td>
<td>v ← Rezd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td></td>
<td></td>
<td>v ← Rezd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td></td>
<td></td>
<td></td>
<td>v ← +Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v ← +Infinity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v ← p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v ← src2</td>
<td></td>
</tr>
</tbody>
</table>

**vxsnan_flag** ← 1

**vxisi_flag** ← 1

**vximz_flag** ← 1

**Explanation:**
- **src1** The quad-precision floating-point value in VSR[VR(A+32)].
- **src2** The quad-precision floating-point value in VSR[VR(T+32)].
- **src3** The quad-precision floating-point value in VSR[VR(B+32)].
- **dQNaN** Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000).
- **NZF** Nonzero finite number.
- **Rezd** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **quiet(x)** Return a QNaN with the payload of x.
- **add(x, y)** Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- **mul(x, y)** Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p** The intermediate product having unbounded range and precision.
- **v** The intermediate result having unbounded range and precision.

**Table 85. Actions for xsmaddpq[o]**
**VSX Scalar Maximum Double-Precision XX3-form**

**xsmaxdp**

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 0 | 60 | T | A | B | 160 | VSR[XA] | 128 |

if MSR.VSX=0 then VSX_Unavailable();

reset_xflags();

src1 ← VSR[32×AX+A].dword[0]

src2 ← VSR[32×BX+B].dword[0]

result ← bfpmu_maxnum(src1, src2)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do

VSR[32×TX+T].dword[0] ← result

VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000

don end

Let XT be the value 32×TX + T.

Let XA be the value 32×AX + A.

Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src1 is greater than src2, src1 is placed into doubleword element 0 of VSR[XT]. Otherwise, src2 is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

The maximum of +0 and -0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN is that SNaN converted to a QNaN.

FPFR, FR and FI are not modified.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified.

See Table 86.

**Special Registers Altered**

| FX | VXSNAN |

**VSR Data Layout for xsmaxdp**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
### Table 86. Actions for xsmaxdp

<table>
<thead>
<tr>
<th>src2</th>
<th>src1</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(src1)</td>
</tr>
<tr>
<td>-NZF</td>
<td>T(src1)</td>
<td>T(M(src1,src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>-Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(M(src1,src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2** The double-precision floating-point value in doubleword element 0 of VSR[XT].
- **NZF** Nonzero finite number.
- **Q(x)** Return a QNaN with the payload of x.
- **M(x,y)** Return the greater of floating-point value x and floating-point value y.
- **T(x)** The value x is placed in doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are set to 0.
- **fx(x)** If x is equal to 0, FX is set to 1. x is set to 1.
- **VXSNAN** Floating-Point Invalid Operation Exception (SNaN) status flag. FPSCR_VXSNAN. If VE=1, update of VSR[XT] is suppressed.
VSX Scalar Maximum Type-C Double-Precision XX3-form

xsmaxcdp XT,XA,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>128</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()}

reset_xflags()}
src1 ← VSR[32×AX+A].dword[0]
src2 ← VSR[32×BX+B].dword[0]
result ← bfp64_MAXIMUM_TYPE_C(src1, src2)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
vex_flag ← FPSCR.VE & vxsnan_flag
if vex_flag=0 then do
  VSR[32×TX+T].dword[0] ← result
  VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
end
Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].
Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.
If either src1 or src2 is a NaN, result is src2.
Otherwise, if src1 is greater than src2, result is src1.
Otherwise, result is src2.
The contents of doubleword 0 of VSR[XT] are set to the value result.
The contents of doubleword 1 of VSR[XT] are set to 0.
If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:
FX VXSNAN

--- Programming Note ---
Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
Table 87. Actions for xsmacd

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>-NZF</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>-Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
</tbody>
</table>

Explanation:

src1 The double-precision floating-point value in doubleword element 0 of VSR[0A].
src2 The double-precision floating-point value in doubleword element 0 of VSR[0T].
NZF Nonzero finite number.
M(x,y) Return the greater of floating-point value x and floating-point value y.
T(x) The value x is placed in doubleword element 0 of VSR[0T] in double-precision format.
The contents of doubleword element 1 of VSR[0T] are set to 0.
FPRF, FR and FI are not modified.
fx(x) If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN Floating-Point Invalid Operation Exception (SNaN) status flag, VXSNAN. If VE=1, update of VSR[0T] is suppressed.

Table 87. Actions for xsmacd

Chapter 7. Vector-Scalar Extension Facility 737
VSX Scalar Maximum Type-C Quad-Precision X-form

<table>
<thead>
<tr>
<th>xsmaxcqp</th>
<th>VRT, VRA, VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Let src1 be the quad-precision floating-point value in VSR[VRA+32].

Let src2 be the quad-precision floating-point value in VSR[VRB+32].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If either src1 or src2 is a NaN, result is src2.

Otherwise, if src1 is greater than src2, result is src1.

Otherwise, result is src2.

The contents of VSR[VRT+32] are set to the value result.

If a trap-enabled Invalid Operation occurs, VSR[VRT+32] is not modified.

Special Registers Altered:

FX VXSNAN

Table 88. Actions for xsmaxcqp

<table>
<thead>
<tr>
<th>src2</th>
<th>+Infinity</th>
<th>+INFINITY</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+INFINITY</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
</tbody>
</table>

Explanation:
src1 The quad-precision floating-point value in VSR[VRA+32].
src2 The quad-precision floating-point value in VSR[VRB+32].
NZF Nonzero finite number.
M(x, y) Return the greater of floating-point value x and floating-point value y.
T(x) The value x is placed in VSR[XT] in quad-precision format.
FPRF, FR and FI are not modified.
fx(x) If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN Floating-Point Invalid Operation Exception (SNaN) status flag. VXSNAN. IFE=1, update of VSR[VRT+32] is suppressed.

Table 88. Actions for xsmaxcqp
### VSX Scalar Maximum Type-J Double-Precision XX3-form

#### xsmaxjdp

<table>
<thead>
<tr>
<th>XT,XA,XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← VSR[32×AX+A].dword[0]
src2 ← VSR[32×BX+B].dword[0]
result ← bfp64_MAXIMUM_TYPE_J(src1,src2)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
  VSR[32×TX+T].dword[0] ← result
  VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
end

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If src1 is a NaN, result is src1.

Otherwise, if src2 is a NaN, result is src2.

Otherwise, if src1 is a Zero and src2 is a Zero and either src1 or src2 is a +Zero, the result is +Zero.

Otherwise, if src1 is a -Zero and src2 is a -Zero, the result is -Zero.

Otherwise, result is src2.

The contents of doubleword 0 of VSR[XT] are set to the value result.

The contents of doubleword 1 of VSR[XT] are set to 0.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

#### Special Registers Altered:

| FX | VXSNAN |

#### Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

#### VSR Data Layout for xsmaxjdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

0x64 127
## Power ISA™ I

### Table 89. Actions for xsmaxjdp

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>T(±MF)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>–NZF</td>
<td>T(src1)</td>
<td>T(M(src1, src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>–Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(±Zero)</td>
<td>T(+Zero)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(±Zero)</td>
<td>T(+Zero)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(M(src1, src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(+MF)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
</tbody>
</table>

### Explanation:

- **src1**: The double-precision floating-point value in doubleword element 0 of `VSR[XA].`
- **src2**: The double-precision floating-point value in doubleword element 0 of `VSR[XT].`
- **NZF**: Nonzero finite number.
- **M(x, y)**: Return the greater of floating-point value x and floating-point value y.
- **T(x)**: The value x is placed in doubleword element 0 of `VSR[XT]` in double-precision format. The contents of doubleword element 1 of `VSR[XT]` are set to 0.
- **FX**: FPRF, FR, and FI are not modified.
- **fx(x)**: If x is equal to 0, FX is set to 1. x is set to 1.
- **VXSNAN**: Floating-Point Invalid Operation Exception (SNaN) status flag, VXSNAN. If VE = 1, update of `VSR[XT]` is suppressed.
**VSX Scalar Minimum Double-Precision XX3-form**

```plaintext
xsmindp XT,XA,XB

let XT be the value 32×TX + T.
let XA be the value 32×AX + A.
let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src1 is less than src2, src1 is placed into doubleword element 0 of VSR[XT] in double-precision format.
Otherwise, src2 is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

The minimum of +0 and –0 is –0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN is that SNaN converted to a QNaN.

FPRF, FR and FI are not modified.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified.

See Table 90.

**Special Registers Altered**

| FX | VXSNAN |

**Programming Note**

This instruction can be used to operate on single-precision source operands.

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(src1)</td>
</tr>
<tr>
<td>–Infinity</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(src1)</td>
</tr>
<tr>
<td>–NZF</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(src1)</td>
</tr>
<tr>
<td>–Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(Q(src2))</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(Q(src2))</td>
<td>T(src1)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
</tbody>
</table>

**Explanation:**

src1 The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2 The double-precision floating-point value in doubleword element 0 of VSR[XT].
NZF Nonzero finite number.
Q(x) Return a QNaN with the payload of x.
M(x,y) Return the lesser of floating-point value x and floating-point value y.
T(x) The value x is placed in doubleword element i (i=0,1) of VSR[XT] in double-precision format.
The contents of doubleword element 1 of VSR[XT] are set to 0.
FPRF, FR and FI are not modified.
fx(x) If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCR\_VXSNAN. If VE=1, update of VSR[XT] is suppressed.

**Table 90. Actions for xsmindp**
**VSX Scalar Minimum Type-C Double-Precision XX3-form**

`xsmincdp` XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>136</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

src1 ← VSR[32×AX+A].dword[0]
src2 ← VSR[32×BX+B].dword[0]
result ← bfp64_MINIMUM_TYPE_C(src1,src2)

if vsxnan_flag=1 then SetFX(FPSCR.VXSNAN)
vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
  VSR[32×TX+T].dword[0] ← result
  VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
end

Let XT be the value 32×T + T.
LetXA be the value 32×A + A.
LetXB be the value 32×B + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].
Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.
If either src1 or src2 is a NaN, result is src2.
Otherwise, if src1 is less than src2, result is src1.
Otherwise, result is src2.

The contents of doubleword 0 of VSR[XT] are set to the value result.
The contents of doubleword 1 of VSR[XT] are set to 0.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

**Special Registers Altered:**
FX VXSNAN

---

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
### Table 91. Actions for xsmincdp

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>–NZF</td>
<td>T(src2)</td>
<td>T(M(src1, src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>–Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(M(src1, src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2** The double-precision floating-point value in doubleword element 0 of VSR[XT].
- **NZF** Nonzero finite number.
- **M(x, y)** Return the lesser of floating-point value x and floating-point value y.
- **T(x)** The value x is placed in doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are set to 0.
- **FX**, **FPRF**, **FR**, and **FI** are not modified.
- **fx(x)** If x is equal to 0, **FX** is set to 1. x is set to 1.
- **VXSNAN** Floating-Point Invalid Operation Exception (SNaN) status flag, **VXSNAN**. If **VE** = 1, update of VSR[XT] is suppressed.
### VSX Scalar Minimum Type-C Quad-Precision X-form

**xsmincqp**

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>740</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable();

src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32]);
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32]);

vxsnan_flag ← (src1.class.SNaN=1) | (src2.class.SNaN=1)

if (src1.class.SNaN=1) | (src1.class.QNaN=1) |
| (src2.class.SNaN=1) | (src2.class.QNaN=1) then
result ← VSR[VRB+32]
else if bfp_COMPARE_LT(src1,src2) then
result ← VSR[VRA+32]
else
result ← VSR[VRB+32]

vex_flag ← FPSCR.VE & vxsnan_flag

if vxsnan_flag=1 then SetFX(VXSNAN)
if vex_flag=0 then
VSR[VRT+32] ← result

Let src1 be the quad-precision floating-point value in VSR[VRA+32].

Let src2 be the quad-precision floating-point value in VSR[VRB+32].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If either src1 or src2 is a NaN, result is src2.

Otherwise, if src1 is less than src2, result is src1.

Otherwise, result is src2.

The contents of VSR[VRT+32] are set to the value result.

If a trap-enabled Invalid Operation occurs, VSR[VRT+32] is not modified.

**Special Registers Altered:**

- FX
- VXSNAN

#### Table 92. Actions for xsmincqp

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>–NZF</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>–Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
</tbody>
</table>

#### Explanation:

- src1 The quad-precision floating-point value in VSR[VRA+32].
- src2 The quad-precision floating-point value in VSR[VRB+32].
- NZF Nonzero finite number.
- M(x,y) Return the lesser of floating-point value x and floating-point value y.
- T(x) The value x is placed in VSR[VRT+32] in quad-precision format.
- FPRF, FR and FI are not modified.
- fx(x) If x is equal to 0, FX is set to 1. x is set to 1.
- VXSNAN Floating-Point Invalid Operation Exception (SNaN) status flag, VXSNAN. If VE=1, update of VSR[VRT+32] is suppressed.
**VSX Scalar Minimum Type-J Double-Precision XX3-form**

**xsminjdp**  
XT,XA,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>152</th>
<th>VSR</th>
<th>0x0000 0x0000 0x0000 0x0000</th>
</tr>
</thead>
</table>

- **if MSR.VSX=0 then** VSX_Unavailable()
- **reset_xflags()**
- **src1** ← VSR[32×AX+A].dword[0]
- **src2** ← VSR[32×BX+B].dword[0]
- **result** ← bfp64_MINIMUM_TYPE_J[src1, src2]
- **if vxsnan_flag=1 then** SetFX(FPSCR.VXSNAN)
- **vex_flag** ← FPSCR.VE & vxsnan_flag
- **if vex_flag=0 then do**
  - VSR[32×TX+T].dword[0] ← result
  - VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
- **end

Let **XT** be the value 32×TX + T.
Let **XA** be the value 32×AX + A.
Let **XB** be the value 32×BX + B.

Let **src1** be the double-precision floating-point value in doubleword 0 of VSR[XA].
Let **src2** be the double-precision floating-point value in doubleword 0 of VSR[XB].

If **src1** or **src2** is a SNaN, an Invalid Operation exception occurs.

If **src1** is a NaN, **result** is **src1**.
Otherwise, if **src2** is a NaN, **result** is **src2**.

Otherwise, if **src1** is a Zero and **src2** is a Zero and either **src1** or **src2** is a -Zero, the result is -Zero.
Otherwise, if **src1** is a +Zero and **src2** is a +Zero, the result is +Zero.
Otherwise, if **src1** is less than **src2**, **result** is **src1**.
Otherwise, **result** is **src2**.

The contents of doubleword 0 of VSR[XT] are set to the value **result**.
The contents of doubleword 1 of VSR[XT] are set to 0.
If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

**Special Registers Altered:**
- FX VXSNAN

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

**VSR Data Layout for xsminjdp**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

0 64 127
### Table 93. Actions for xsminjdp

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(-Zero)</td>
<td>T(-Zero)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(-Zero)</td>
<td>T(+Zero)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(+INF)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2** The double-precision floating-point value in doubleword element 0 of VSR[XT].
- **NZF** Nonzero finite number.
- **M(x,y)** Return the greater of floating-point value x and floating-point value y.
- **T(x)** The value x is placed in doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are set to 0.
- **FX**, **FR**, and **FI** are not modified.
- **fx(x)** If x is equal to 0, FX is set to 1. x is set to 1.
- **VXSNAN** Floating-Point Invalid Operation Exception (SNaN) status flag, VXSNAN. If VE=1, update of VSR[XT] is suppressed.
VSX Scalar Multiply-Subtract Type-A
Double-Precision XX3-form

xsmsubadp

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>49</th>
<th>VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

VSX Scalar Multiply-Subtract Type-M
Double-Precision XX3-form

xsmsubmdp

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>57</th>
<th>VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

if "xsmsubadp" then do
src1 ← bfp_CONVERT_FROM_BFP128(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[32×TX+T].dword[0])
src3 ← bfp_CONVERT_FROM_BFP128(VSR[32×BX+B].dword[0])
end

if "xsmsubmdp" then do
src1 ← bfp_CONVERT_FROM_BFP128(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[32×BX+B].dword[0])
src3 ← bfp_CONVERT_FROM_BFP128(VSR[32×TX+T].dword[0])
end

v ← bfp_MULTIPLY_ADD(src1, src3, bfp_NEGATE(src2))
rnd ← bfp_ROUND_TO_BFP64(0b0, FPSCR.RN, v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← Fprf_CLASS_BFP64(result)
FPSCR.FR ← vex_flag
FPSCR.FI ← xx_flag
end else do
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
end

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

Special Registers Altered

\[
\begin{array}{lllllllll}
FPRF & FR & FI & FX & OX & UX & XX & VXSNAN & VXISI & VXIMZ
\end{array}
\]

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

### VSR Data Layout for xsmsubadp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XT].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

### VSR Data Layout for xsmsubmdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XT].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>
### Table 94. Actions for `xmsub(a|m)dp`

<table>
<thead>
<tr>
<th>src</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>p ← dQNaN</td>
<td>p ← –Infinity</td>
<td>p ← –Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
</tr>
</tbody>
</table>

#### Explanation:

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: For `xmsubdp`, the double-precision floating-point value in doubleword element 0 of VSR[XT].
- **src3**: For `xmsubdp`, the double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)**: Return a QNaN with the payload of x.
- **S(x,y)**: Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
  - Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
- **M(x,y)**: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p**: The intermediate product having unbounded range and precision.
- **v**: The intermediate result having unbounded range and precision.
VSX Scalar Multiply-Subtract Type-A
Single-Precision XX3-form

\[
\text{xsmsubasp} \rightarrow XT, XA, XB
\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>17</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VSX Scalar Multiply-Subtract Type-M
Single-Precision XX3-form

\[
\text{xsmsubmsp} \rightarrow XT, XA, XB
\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

if "xsmsubasp" then do
src1 ← bfp_CONVERT_FROM_BFP128(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[32×TX+T].dword[0])
src3 ← bfp_CONVERT_FROM_BFP128(VSR[32×BX+B].dword[0])
end

if "xsmsubmsp" then do
src1 ← bfp_CONVERT_FROM_BFP128(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[32×BX+B].dword[0])
src3 ← bfp_CONVERT_FROM_BFP128(VSR[32×TX+T].dword[0])
end

v ← bfp_MULTIPLY_ADD(src1, src3, bfp_NEGATE(src2))
rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN, v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & xx_flag

if vex_flag=0 then do
VSR[32×TX+T].dword[0] ← result64
VSR[32×TX+T].dword[1] ← 0b0000_0000_0000_0000
FPSCR.FPRF ← pref_CLASS_BFP32(result128)
FPSCR.FR ← inc_flag
FPSCR.FI ← xx_flag
end
else do
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
end

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For \text{xsmsubasp}, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[AX].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[BX].

For \text{xsmsubmsp}, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[AX].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[BX].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied\(^{1}\) by src3, producing a product having unbounded range and precision.

See part 1 of Table 95, “Actions for xsmsub(a|m)sp”.

src2 is negated and added\(^{2}\) to the product, producing a sum having unbounded range and precision.

The result, having unbounded range and precision, is normalized\(^{3}\).

See part 2 of Table 95, “Actions for xsmsub(a|m)sp”.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.


The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

Special Registers Altered

| PPRF | FR | FI |
| FX | OX | UX | XX | VXSNAN | VXISI | VXIMZ |

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

**VSR Data Layout for xsmsubasp**

```
src1  VSR[XA].dword[0]  unused
src2  VSR[XT].dword[0]  unused
src3  VSR[XB].dword[0]  unused
tgt  VSR[XB].dword[0]  0x0000_0000_0000_0000

```

**VSR Data Layout for xsmsubmsp**

```
src1  VSR[XA].dword[0]  unused
src2  VSR[XB].dword[0]  unused
src3  VSR[XT].dword[0]  unused
tgt  VSR[XB].dword[0]  0x0000_0000_0000_0000
```
### Table 95. Actions for `xsmsub(a|m)sp`

<table>
<thead>
<tr>
<th>Part 1: Multiply</th>
<th><code>src1</code>: The double-precision floating-point value in doubleword element 0 of VSR[XA].</th>
<th><code>src2</code>: For <code>xsmsubasp</code>, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For <code>xsmsubbsp</code>, the double-precision floating-point value in doubleword element 0 of VSR[XB].</th>
<th><code>src3</code>: For <code>xsmsubasp</code>, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For <code>xsmsubbsp</code>, the double-precision floating-point value in doubleword element 0 of VSR[XT].</th>
<th><code>dQNaN</code>: Default quiet NaN (0x7FF8_0000_0000_0000).</th>
<th><code>NZF</code>: Nonzero finite number.</th>
<th><code>Rezd</code>: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.</th>
<th><code>Q(x)</code>: Return a QNaN with the payload of x.</th>
<th><code>S(x,y)</code>: Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).</th>
<th><code>M(x,y)</code>: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.</th>
<th><code>p</code>: The intermediate product having unbounded range and precision.</th>
<th><code>v</code>: The intermediate result having unbounded range and precision.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>–Infinity</strong></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
</tr>
<tr>
<td><strong>–NZF</strong></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
</tr>
<tr>
<td><strong>–Zero</strong></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
</tr>
<tr>
<td><strong>+Zero</strong></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
</tr>
<tr>
<td><strong>+NZF</strong></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
</tr>
<tr>
<td><strong>+Infinity</strong></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← +Infinity</code></td>
</tr>
<tr>
<td><strong>QNaN</strong></td>
<td><code>p ← src1</code></td>
<td><code>p ← src1</code></td>
<td><code>p ← src1</code></td>
<td><code>p ← src1</code></td>
<td><code>p ← src1</code></td>
<td><code>p ← src1</code></td>
<td><code>p ← src1</code></td>
<td><code>p ← src1</code></td>
<td><code>p ← src1</code></td>
<td><code>p ← src1</code></td>
<td><code>p ← src1</code></td>
</tr>
<tr>
<td><strong>NaN</strong></td>
<td><code>p ← Q(src1)</code></td>
<td><code>v ← Q(src1)</code></td>
<td><code>v ← Q(src1)</code></td>
<td><code>v ← Q(src1)</code></td>
<td><code>v ← Q(src1)</code></td>
<td><code>v ← Q(src1)</code></td>
<td><code>v ← Q(src1)</code></td>
<td><code>v ← Q(src1)</code></td>
<td><code>v ← Q(src1)</code></td>
<td><code>v ← Q(src1)</code></td>
<td><code>v ← Q(src1)</code></td>
</tr>
</tbody>
</table>

**Part 2: Subtract**

| **–Infinity** | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` |
| **–NZF** | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` |
| **–Zero** | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` |
| **+Zero** | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` |
| **+NZF** | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` |
| **+Infinity** | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` | `v ← +Infinity` |
| **QNaN & src1 is a NaN** | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` |
| **QNaN & src1 not a NaN** | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` | `v ← p` |

**Explanation:**

- `src1`: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- `src2`: For `xsmsubasp`, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For `xsmsubbsp`, the double-precision floating-point value in doubleword element 0 of VSR[XB].
- `src3`: For `xsmsubasp`, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For `xsmsubbsp`, the double-precision floating-point value in doubleword element 0 of VSR[XT].
- `dQNaN`: Default quiet NaN (0x7FF8_0000_0000_0000).
- `NZF`: Nonzero finite number.
- `Rezd`: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- `Q(x)`: Return a QNaN with the payload of x.
- `S(x,y)`: Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
- `M(x,y)`: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- `p`: The intermediate product having unbounded range and precision.
- `v`: The intermediate result having unbounded range and precision.
VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd] X-form

\[
x\text{msubqp} \quad \text{VRT, VRA, VRB} \quad (\text{RO} = 0) \\
x\text{msubqpo} \quad \text{VRT, VRA, VRB} \quad (\text{RO} = 1)
\]

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>420</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRT+32])
src3 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

v ← bfp_MULTIPLY_ADD(src1, src3, bfp_NEGATE(src2))
rnd ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
  VSR[VRT+32] ← result
  FPSCR.FPRF ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag

Let \( src1 \) be the floating-point value in \( VSR[VRA+32] \) represented in quad-precision format.

Let \( src2 \) be the floating-point value in \( VSR[VRT+32] \) represented in quad-precision format.

Let \( src3 \) be the floating-point value in \( VSR[VRB+32] \) represented in quad-precision format.

If either \( src1 \), \( src2 \), or \( src3 \) is a Signalling NaN, an Invalid Operation exception occurs and \( VXSNAN \) is set to 1.

If \( src1 \) is an Infinity value and \( src3 \) is a Zero value, or if \( src1 \) is a Zero value and \( src3 \) is an Infinity value, an Invalid Operation exception occurs and \( VXIMZ \) is set to 1.

If \( src2 \) and the product of \( src1 \) and \( src3 \) are Infinity values having same signs, an Invalid Operation exception occurs and \( VXISI \) is set to 1.

If \( src1 \) is a Signalling NaN, the result is the Quiet NaN corresponding to \( src1 \).

Otherwise, if \( src1 \) is a Quiet NaN, the result is \( src1 \).

Otherwise, if \( src2 \) is a Signalling NaN, the result is the Quiet NaN corresponding to \( src2 \).

Otherwise, if \( src3 \) is a Quiet NaN, the result is \( src3 \).

Otherwise, if \( src1 \) is an Infinity value and \( src3 \) is a Zero value, or if \( src1 \) is a Zero value and \( src3 \) is an Infinity value, the default Quiet NaN is set.

Otherwise, if the product of \( src1 \) and \( src3 \), and \( src2 \) are Infinity values having same signs, the result is the default Quiet NaN.

Otherwise, do the following.

1. \( src1 \) is multiplied by \( src3 \), producing a product having unbounded significand precision and exponent range.

2. \( src2 \) is negated and added to the product, producing a sum having unbounded range and precision.

See part 1 of Table 96. "Actions for xmsubqpo[0]."

See part 2 of Table 96. "Actions for xmsubqpo[0]."

If the intermediate result is \( \text{Tiny} \) (i.e., the unbiased exponent is less than \(-16382\) and \( UE=0 \), the significand is shifted right \( N \) bits, where \( N \) is the difference between \(-16382\) and the unbiased exponent of the intermediate result). The exponent of the intermediate result is set to the value \(-16382\).

If \( RO=1 \), let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by \( RN \). Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.


The result is placed into \( VSR[VRT+32] \) in quad-precision format.

FPFR is set to the class and sign of the result. \( FR \) is set to indicate if the rounded result was incremented. \( FI \) is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, \( FR \) and \( FI \) are set to 0.

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

**Special Registers Altered:**

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX</td>
<td>VXSNAN</td>
<td>VXIMZ</td>
</tr>
</tbody>
</table>

**VSR Data Layout for xsmsubqp[o]**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

| 0 | 127 |
### Part 1: Multiply

<table>
<thead>
<tr>
<th>src3</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>p + Infinity</td>
<td>p + QNaN</td>
<td>viximz_flag = 1</td>
<td>p + Infinity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td></td>
<td>p + QNaN</td>
<td>viximz_flag = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–Zero</td>
<td>p + QNaN</td>
<td></td>
<td>viximz_flag = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td></td>
<td>p + QNaN</td>
<td>viximz_flag = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td></td>
<td>p + QNaN</td>
<td>viximz_flag = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td></td>
<td>p + QNaN</td>
<td>viximz_flag = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td></td>
<td>p + QNaN</td>
<td>viximz_flag = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td></td>
<td>p + QNaN</td>
<td>viximz_flag = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Part 2: Subtract

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td></td>
<td></td>
<td>v ← –Infinity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td></td>
<td>v ← sub(p,src2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–Zero</td>
<td></td>
<td>v ← Rezd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td></td>
<td>v ← +Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td></td>
<td>v ← sub(p,src2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td></td>
<td>v ← +Infinity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>v ← QNaN</td>
<td>vixis_i_flag = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← src2</td>
<td>vixis_i_flag = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**
- **src1** The quad-precision floating-point value in VSR[VRA+32].
- **src2** The quad-precision floating-point value in VSR[VRT+32].
- **src3** The quad-precision floating-point value in VSR[VRB+32].
- **QNaN** Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000_0000).
- **NZF** Nonzero finite number.
- **Rezd** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **quiet(x)** Return a QNaN with the payload of x.
- **sub(x,y)** Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.

**Note:** If x = y, v is considered to be an exact-zero-difference result (Rezd).
- **mul(x,y)** Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p** The intermediate product having unbounded range and precision.
- **v** The intermediate result having unbounded range and precision.

---

Table 96. Actions for xsmsubqp[o]
VSX Scalar Multiply Double-Precision XX3-form

\[
\text{xxsmuldp} \quad \text{XT,XA,XB}
\]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v ← bfp_MULTIPLY(src1, src2)
red ← bfp_ROUND_TO_BFP64(0b0, FPSCR.RN, v)
result ← bfp64_CONVERT_FROM_BFP(red)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vximz_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
  VSR[32×TX+T].dword[0] ← result
  VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
  FPSCR.FPRF ← fprf_CLASS_BFP64(result)
  FPSCR.FR ← inc_flag
  FPSCR.FI ← xx_flag
end else do
  FPSCR.FR ← 0b0
  FPSCR.FI ← 0b0
end

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is multiplied[1] by src2, producing a product having unbounded range and precision.

The product is normalized[2].

See Table 97.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.


The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

Special Registers Altered

FPRF FR FI
FX OX UX XX VXSNAN VXIMZ

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>vximz_flag ← 1</td>
<td>v ← +Infinity</td>
<td>vximz_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← M(src1, src2)</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← M(src1, src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vximz_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← dQNaN</td>
<td>vximz_flag ← 1</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← dQNaN</td>
<td>vximz_flag ← 1</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← dQNaN</td>
<td>vximz_flag ← 1</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>vximz_flag ← 1</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity</td>
<td>v ← M(src1, src2)</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← M(src1, src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vximz_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← -Infinity</td>
<td>p ← -Infinity</td>
<td>v ← dQNaN</td>
<td>vximz_flag ← 1</td>
<td>v ← dQNaN</td>
<td>vximz_flag ← 1</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vximz_flag ← 1</td>
<td>v ← Q(src1) vximz_flag ← 1</td>
<td>v ← Q(src1) vximz_flag ← 1</td>
<td>v ← Q(src1) vximz_flag ← 1</td>
<td>v ← Q(src1) vximz_flag ← 1</td>
<td>v ← Q(src1) vximz_flag ← 1</td>
<td>v ← Q(src1) vximz_flag ← 1</td>
<td>v ← Q(src1) vximz_flag ← 1</td>
</tr>
</tbody>
</table>

Explanation:
- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **M(x, y)**: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **Q(x)**: Return a QNaN with the payload of x.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.

Table 97. Actions for xsmuldp
VSX Scalar Multiply Quad-Precision [using round to Odd] X-form

\[
xsmulp \quad \text{VRT, VRA, VRB} \quad (RO=0)
xsmulpo \quad \text{VRT, VRA, VRB} \quad (RO=1)
\]

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>36</th>
<th>RO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

\[
\text{if MSR.VSX=0 then VSX_Unavailable()}
\]
\[
\text{reset_xflags()}
\]
\[
\text{src1 } \leftarrow \text{bfp\_CONVERT\_FROM\_BFP128(VSR[VRA+32])}
\]
\[
\text{src2 } \leftarrow \text{bfp\_CONVERT\_FROM\_BFP128(VSR[VRB+32])}
\]
\[
v \leftarrow \text{bfp\_MULTIPLY(src1, src2)}
\]
\[
rnd \leftarrow \text{bfp\_ROUND\_TO\_BFP128(RO, FPSCR.RN, v)}
\]
\[
\text{result } \leftarrow \text{bfp128\_CONVERT\_FROM\_BFP(rnd)}
\]
\[
\text{if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)}
\]
\[
\text{if vximz_flag=1 then SetFX(FPSCR.VXIMZ)}
\]
\[
\text{if ox_flag=1 then SetFX(FPSCR.OX)}
\]
\[
\text{if ux_flag=1 then SetFX(FPSCR.UX)}
\]
\[
\text{if xx_flag=1 then SetFX(FPSCR.XX)}
\]
\[
\text{vx_flag } \leftarrow \text{vxsnan_flag} \mid \text{vximz_flag}
\]
\[
\text{vex_flag } \leftarrow \text{FPSCR.VE} \& \text{vx_flag}
\]
\[
\text{if vx_flag=0 then do}
\]
\[
\text{VSR[VRT+32] } \leftarrow \text{result}
\]
\[
\text{FPSCR.FPRF } \leftarrow \text{fprf\_CLASS\_BFP128(result)}
\]
\[
\text{end}
\]
\[
\text{FPSCR.FR } \leftarrow \text{if vx_flag=0} \& \text{inc_flag}
\]
\[
\text{FPSCR.FI } \leftarrow \text{if vx_flag=0} \& \text{xx_flag}
\]

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 is an Infinity value and src2 is a Zero value, or if src1 is a Zero value and src2 is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src1 is an Infinity value and src2 is a Zero value, or if src1 is a Zero value and src2 is an Infinity value, the result is the default Quiet NaN.

Otherwise, do the following.

The normalized product of src1 multiplied by src2 is produced with unbounded significand precision and exponent range.

See Table 98, "Actions for xsmulqp[o]".

If the intermediate result is Tiny (i.e., the unbiased exponent is less than \(-16382\)) and UE=0, the significand is shifted right \(N\) bits, where \(N\) is the difference between \(-16382\) and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value \(-16382\).

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.


The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result," on page 662.

Special Registers Altered:

\[
\text{FPRF FR FI FX VXSNAN VXIMZ OX UX XX}
\]

1. The quad-precision default Quiet NaN is the value, \(0x7FFF\_8000\_0000\_0000\_0000\_0000\_0000\).
### VSR Data Layout for xsmulqp[o]

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[ VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[ VRB+32]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[ VRT+32]</td>
</tr>
</tbody>
</table>

#### Table 98. Actions for xsmulqp[o]

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>x &lt;- Infinity</td>
<td>x &lt;- QNaN</td>
<td>x &lt;- SignFlag = 1</td>
<td>v &lt;- QNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>v &lt;- mul(src1, src2)</td>
<td>v &lt;- mul(src1, src2)</td>
<td>v &lt;- mul(src1, src2)</td>
<td>v &lt;- mul(src1, src2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>x &lt;- QNaN</td>
<td>v &lt;- Zero</td>
<td>v &lt;- Zero</td>
<td>v &lt;- QNaN</td>
<td>v &lt;- QNaN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v &lt;- mul(src1, src2)</td>
<td>v &lt;- Zero</td>
<td>v &lt;- Zero</td>
<td>v &lt;- QNaN</td>
<td>v &lt;- QNaN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v &lt;- mul(src1, src2)</td>
<td>v &lt;- mul(src1, src2)</td>
<td>v &lt;- mul(src1, src2)</td>
<td>v &lt;- mul(src1, src2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>x &lt;- Infinity</td>
<td>x &lt;- QNaN</td>
<td>x &lt;- SignFlag = 1</td>
<td>v &lt;- QNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>x &lt;- src1</td>
<td>v &lt;- QNaN</td>
<td>v &lt;- QNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>v &lt;- QNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Explanation:
- **src1**: The quad-precision floating-point value in VSR[ VRA+32].
- **src2**: The quad-precision floating-point value in VSR[ VRB+32].
- **QNaN**: Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **mul(x, y)**: The floating-point value x is multiplied by the floating-point value y. Return the normalized product, having unbounded significand precision and exponent range.
- **quiet(x)**: Convert x to the corresponding Quiet NaN.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
VSX Scalar Multiply Single-Precision
XX3-form

xsmulp XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>11</th>
<th>A</th>
<th>16</th>
<th>B</th>
<th>21</th>
<th>16</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v ← bfp_MULTIPLY(src1,src2)
rv ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rv)
result64 ← bfp64_CONVERT_FROM_BFP(rv)

if vsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vsnan_flag | vximz_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
VSR[32×TX+T].dword[0] ← result64
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP32(result128)
FPSCR.FR ← inc_flag
FPSCR.FI ← xx_flag
end
else do
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
end

Let XT be the value 32×TX + T.
LetXA be the value 32×AX + A.
LetXB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is multiplied\(^1\) by src2, producing a product having unbounded range and precision.

The product is normalized\(^2\).

See Table 99, “Actions for xsmulp,” on page 762.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.


The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

Special Registers Altered
FPRF FR FI
FX OX UX XX VXSNAN VXIMZ

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmulp

| src1 | VSR[XA].dword[0] | unused |
| src2 | VSR[XB].dword[0] | unused |
| tgt  | VSR[XB].dword[0] | 0x0000_0000_0000_0000 |

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 99. Actions for xsmulp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← −Infinity</td>
<td>v ← −Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← M(src1, src2)</td>
<td>v ← +Zero</td>
<td>v ← −Zero</td>
<td>v ← −M(src1, src2)</td>
<td>v ← −Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← −Infinity</td>
<td>v ← M(src1, src2)</td>
<td>v ← −Zero</td>
<td>v ← −Zero</td>
<td>v ← −M(src1, src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← −Infinity</td>
<td>v ← M(src1, src2)</td>
<td>v ← −Zero</td>
<td>v ← −Zero</td>
<td>v ← −M(src1, src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← −Infinity</td>
<td>v ← M(src1, src2)</td>
<td>v ← −Zero</td>
<td>v ← −Zero</td>
<td>v ← −M(src1, src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

#### Explanation:

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **M(x,y)**: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **Q(x)**: Return a QNaN with the payload of x.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.


**VSX Scalar Negative Absolute Double-Precision XX2-form**

xsnabsdp XT,XB

if MSR.VSX=0 then VSX_Unavailable()

src ← VSR[32×BX+B].dword[0]
VSR[32×TX+T].dword[0] ← bfp64_NEGATIVE_ABSOLUTE(src)
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of doubleword element 0 of VSR[XB], with bit 0 set to 1, is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

**Special Registers Altered**

None

**Programming Note**

This instruction can be used to operate on a single-precision source operand.

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

**VSR Data Layout for xsnabsdp**

```
src | VSR[XB].dword[0] | unused
---|------------------|-------
tgt | VSR[XT].dword[0] | 6x0000_0000_0000_0000
```

**VSR Data Layout for xsnabsqp**

```
src | VSR[VRB+32]
---|-------
tgt | VSR[VRT+32]
```

---

**VSX Scalar Negative Absolute Quad-Precision X-form**

xsnabsqp VRT,VRB

if MSR.VSX=0 then VSX_Unavailable()

VSR[VRT+32] ← bfp128_NEGATIVE_ABSOLUTE(VSR[VRB+32])

Let src be the floating-point value in VSR[VRB+32] represented in quad-precision format.

The negative absolute value of src is placed into VSR[VRT+32] in quad-precision format.

**Special Registers Altered:**

None

---

This instruction can be used to operate on a single-precision source operand.

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
VSX Scalar Negate Double-Precision  
**XX2-form**

```
xsnegdp  XT,XB

if MSR.VSX=0 then VSX_Unavailable()||

src ← VSR[32×BX+B].dword[0]
VSR[32×TX+T].dword[0] ← bf64_NEGATE(src)
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of doubleword element 0 of VSR[XB], with
bit 0 complemented, is placed into doubleword element 0 of VSR[XT] .

The contents of doubleword element 1 of VSR[XT] are
set to 0.

**Special Registers Altered**
None
```

---

Programming Note

This instruction can be used to operate on a
single-precision source operand.

---

Programming Note

Previous versions of the architecture allowed the
contents of doubleword 1 of the result register to be
undefined. However, all processors that support
this instruction write 0s into doubleword 1 of the
result register, as is required by this version of the
architecture.

---

**VSR Data Layout for xsnegdp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

---

**VSR Data Layout for xsnegqp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XA \) be the value \( 32 \times AX + A \).
Let \( XB \) be the value \( 32 \times BX + B \).

Let \( src1 \) be the double-precision floating-point value in doubleword element 0 of \( VSR[XA] \).

For \textit{xsnmaddadp}, do the following.
- Let \( src2 \) be the double-precision floating-point value in doubleword element 0 of \( VSR[XT] \).
- Let \( src3 \) be the double-precision floating-point value in doubleword element 0 of \( VSR[XB] \).

For \textit{xsnmaddmdp}, do the following.
- Let \( src2 \) be the double-precision floating-point value in doubleword element 0 of \( VSR[XB] \).
- Let \( src3 \) be the double-precision floating-point value in doubleword element 0 of \( VSR[XT] \).

\( src1 \) is multiplied\(^1\) by \( src3 \), producing a product having unbounded range and precision.

See part 1 of Table 100.

\( src2 \) is added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 100.

The intermediate result is rounded to double-precision using the rounding mode specified by \( RN \).


The result is negated and placed into doubleword element 0 of \( VSR[XT] \) in double-precision format.

The contents of doubleword element 1 of \( VSR[XT] \) are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, \( VSR[XT] \) and FPRF are not modified, and FR and FI are set to 0.

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
See Table 101, “Scalar Floating-Point Final Result with Negation,” on page 768.

Special Registers Altered

<table>
<thead>
<tr>
<th>PPRF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX</td>
<td>UX</td>
<td>XX</td>
</tr>
</tbody>
</table>

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsnmaddadp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XT].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

VSR Data Layout for xsnmaddmdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XT].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>
### Chapter 7. Vector-Scalar Extension Facility

#### Table 100. Actions for xsnmadd(a|m)dp

<table>
<thead>
<tr>
<th>src1</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>-NZF</td>
<td>p ← +Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>-Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Zero</td>
<td>p ← -Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← -Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← -Infinity</td>
<td>p ← dQNaN</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
</tr>
</tbody>
</table>

#### Part 1: Multiply

<table>
<thead>
<tr>
<th>src3</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>-NZF</td>
<td>p ← +Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>-Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Zero</td>
<td>p ← -Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← -Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← -Infinity</td>
<td>p ← dQNaN</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← Q(src3)</td>
<td>p ← Q(src3)</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
</tr>
</tbody>
</table>

#### Part 2: Add

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
</tbody>
</table>

#### Explanation:

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: For **xsnmaddadp**, the double-precision floating-point value in doubleword element 0 of VSR[XT].
- **src3**: For **xsnmadddp**, the double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)**: Return a QNaN with the payload of x.
- **A(x,y)**: Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
- **M(x,y)**: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p**: The intermediate product having unbounded range and precision.
- **v**: The intermediate result having unbounded range and precision.
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>Case</th>
<th>FPSCR.VE</th>
<th>FPSCR.OE</th>
<th>FPSCR.ZE</th>
<th>FPSCR.XE</th>
<th>FPRF flag</th>
<th>FI flag</th>
<th>FR flag</th>
<th>Is r inexact?</th>
<th>Is r incremented?</th>
<th>Is q inexact?</th>
<th>Is q incremented?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special</td>
<td>- - - - - 0 0 0 - - -</td>
<td>T(r), fprf(class(r)), fi(0), fr(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 - - - - - - 0 0 0 - -</td>
<td>T(r), fprf(class(r)), fi(0), fr(0), fx(VXISI)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 - - - - 0 1 - - - - -</td>
<td>T(r), fprf(class(r)), fi(0), fr(0), fx(VXIMZ)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 - - - - - 0 0 1 - - -</td>
<td>T(r), fprf(class(r)), fi(0), fr(0), fx(VXSNAN)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 - - - - 1 1 - - - - -</td>
<td>T(r), fprf(class(r)), fi(0), fr(0), fx(VXSNAN), fx(VXIMZ)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 - - - - - 1 0 0 - -</td>
<td>fx(VXSNAN), error()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 - - - - - - 1 1 1 -</td>
<td>fx(VXSNAN), fx(VXIMZ), error()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Normal</td>
<td>- - - - - - - - - - - - - - - -</td>
<td>T(n(r)), fprf(class(n(r))), fi(0), fr(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- - - - - - - - - - - - - - - -</td>
<td>T(n(r)), fprf(class(n(r))), fi(1), fr(0), fx(XX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- - - - - - - - - - - - - - - -</td>
<td>T(n(r)), fprf(class(n(r))), fi(1), fr(1), fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- - - - - - - - - - - - - - - -</td>
<td>T(n(r)), fprf(class(n(r))), fi(1), fr(1), fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- - - - - - - - - - - - - - - -</td>
<td>T(n(r)), fprf(class(n(r))), fi(1), fr(1), fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- - - - - - - - - - - - - - - -</td>
<td>T(n(r)), fprf(class(n(r))), fi(1), fr(1), fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overflow</td>
<td>- - - - - - - - - - - - - - - -</td>
<td>T(n(q)÷β), fprf(class(n(q)÷β)), fi(0), fr(0), fx(OX), fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- - - - - - - - - - - - - - - -</td>
<td>T(n(q)÷β), fprf(class(n(q)÷β)), fi(1), fr(0), fx(OX), fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- - - - - - - - - - - - - - - -</td>
<td>T(n(q)÷β), fprf(class(n(q)÷β)), fi(1), fr(1), fx(OX), fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Explanation:
- The results do not depend on this condition.
- **class(x)** Classifies the floating-point value \(x\) as defined in Table 2, “Floating-Point Result Flags,” on page 505.
- **error()** The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
- **fi(x)** FPSCR.FI is set to the value \(x\).
- **fprf(x)** FPSCR.FPRF is set to the 5-bit value \(x\).
- **fr(x)** FPSCR.FR is set to the value \(x\).
- **fx(x)** FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
- \(\beta\) Wrap adjust, where \(\beta = 2^{1536}\) for double-precision and \(\beta = 2^{192}\) for single-precision.
- **q** The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target precision, unbounded exponent range.
- **r** The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target precision, bounded exponent range.
- **v** The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.
- **n(x)** The value \(x\) is is negated by complementing the sign bit of \(x\).
- **T(x)** The value \(x\) is placed in element 0 of VSR[XT] in the target precision format.

Table 101: Scalar Floating-Point Final Result with Negation
### Returned Results and Status Setting

| Case | FPSCR.VE | FPSCR.OE | FPSCR.UE | FPSCR.ZE | FPSCR.XE | Is r inexact? (r ≠ v) | Is q inexact? (q ≠ v) | Is r incremented? (|r| > |v|) | Is q incremented? (|q| > |v|) |
|------|---------|---------|---------|---------|---------|---------------------|---------------------|---------------------|---------------------|
| Tiny | – – 0 – – – – no – – | T(n(r)), fprf(class(n(r))), fi(0), fr(0) |
| | – – 0 0 – – – yes no – | T(n(r)), fprf(class(n(r))), fi(1), fr(0), fx(UX), fx(XX) |
| | – – 0 1 – – – yes yes – | T(n(r)), fprf(class(n(r))), fi(1), fr(1), fx(UX), fx(XX) |
| | – – 1 – – – – yes no – | T(n(q)×β), fprf(class(n(q)×β)), fi(0), fr(0), fx(UX), error() |
| | – – 1 – – – – yes yes – | T(n(q)×β), fprf(class(n(q)×β)), fi(1), fr(0), fx(UX), fx(XX), error() |

**Explanation:**

- The results do not depend on this condition.
- **class(x)**: Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 505.
- **error()**: The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
- **fi(x)**: FPSCR.FI is set to the value x.
- **fprf(x)**: FPSCR.FPRF is set to the 5-bit value x.
- **fr(x)**: FPSCR.FR is set to the value x.
- **fx(x)**: FPSCR.FX is set to 1 if FPSCR.x=0; FPSCR.x is set to 1.
- **β**: Wrap adjust, where β = 2\(^{31536}\) for double-precision and β = 2\(^{3192}\) for single-precision.
- **q**: The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target precision, unbounded exponent range.
- **r**: The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target precision, bounded exponent range.
- **v**: The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.
- **n(x)**: The value i is negated by complementing the sign bit of i.
- **T(x)**: The value i is placed in element 0 of VSR[XT] in the target precision format.

The contents of the remaining element(s) of VSR[XT] are set to 0.

Table 101. Scalar Floating-Point Final Result with Negation (Continued)
VSX Scalar Negative Multiply-Add Type-A
Single-Precision XX3-form

\[ \text{xsnmaddsp \ XT,XA,XB} \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>129</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

VSX Scalar Negative Multiply-Add Type-M
Single-Precision XX3-form

\[ \text{xsnmaddmsp \ XT,XA,XB} \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>137</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable();
reset_xflags();

if "xsnmaddsp" then do
    src1 ← bfp_CONVERT_FROM_BFP128(VSR[32×AX+A].dword[0])
    src2 ← bfp_CONVERT_FROM_BFP128(VSR[32×TX+T].dword[0])
    src3 ← bfp_CONVERT_FROM_BFP128(VSR[32×BX+B].dword[0])
end

if "xsnmaddmsp" then do
    src1 ← bfp_CONVERT_FROM_BFP128(VSR[32×AX+A].dword[0])
    src2 ← bfp_CONVERT_FROM_BFP128(VSR[32×BX+B].dword[0])
    src3 ← bfp_CONVERT_FROM_BFP128(VSR[32×TX+T].dword[0])
end

v ← bfp_MULTIPLY_ADD(src1, src3, src2)

rnd ← bfp_NEGATE(bfp_ROUND_TO_BFP32(FPSCR.RN, v))

result32 ← bfp32_CONVERT_FROM_BFP(rnd)

result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
    VSR[32×TX+T].dword[0] ← result64
    VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
    FPSCR.FPRF ← (fpf_CLASS_BFP32(result32))
    FPSCR.FR ← inc_flag
    FPSCR.FI ← xx_flag
end
else do
    FPSCR.FR ← 0b0
    FPSCR.FI ← 0b0
end
See Table 101, “Scalar Floating-Point Final Result with Negation,” on page 768.

Special Registers Altered

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX</td>
<td>OX</td>
<td>UX</td>
</tr>
</tbody>
</table>

| VXSNAN | VXISI | VXIMZ |

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

### VSR Data Layout for xsnmaddasp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XT].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

| 0 | 64 | 127 |

### VSR Data Layout for xsnmaddmsp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XT].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

| 0 | 64 | 127 |
### Table 102. Actions for \texttt{xsnmadd(a|m)sp}

<table>
<thead>
<tr>
<th>src2</th>
<th>(~\infty)</th>
<th>(-\infty)</th>
<th>(-\text{NZF})</th>
<th>(-\text{Zero})</th>
<th>(+\text{Zero})</th>
<th>(+\text{NZF})</th>
<th>(+\infty)</th>
<th>\text{QNaN}</th>
<th>\text{SNaN}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\text{dQNaN})</td>
<td>(+\text{dQNaN})</td>
<td>(-\infty)</td>
<td>(-\infty)</td>
<td>(+\text{QNaN})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(-\text{NZF})</td>
<td>(+\text{src1})</td>
<td>(+\text{src1})</td>
<td>(-\text{NZF})</td>
<td>(-\text{NZF})</td>
<td>(+\text{SNaN})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{src1})</td>
<td>(+\text{src1})</td>
<td>(-\text{Zero})</td>
<td>(-\text{Zero})</td>
<td>(+\text{SNaN})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+\text{Zero})</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{src1})</td>
<td>(+\text{src1})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{QNaN})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{src1})</td>
<td>(+\text{src1})</td>
<td>(-\text{NZF})</td>
<td>(-\text{NZF})</td>
<td>(+\text{QNaN})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\text{NZF})</td>
<td>(+\text{src1})</td>
<td>(+\text{src1})</td>
<td>(+\text{Infty})</td>
<td>(+\text{Infty})</td>
<td>(+\text{QNaN})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{QNaN}</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{src1})</td>
<td>(+\text{src1})</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{QNaN})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{SNaN}</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{src1})</td>
<td>(+\text{src1})</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{QNaN})</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Explanation:

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: For \texttt{xsnmaddsp}, the double-precision floating-point value in doubleword element 0 of VSR[XT].
- **src3**: For \texttt{xsnmaddmsp}, the double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FFE_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)**: Return a QNaN with the payload of x.
- **A(x,y)**: Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
  
  Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- **M(x,y)**: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p**: The intermediate product having unbounded range and precision.
- **v**: The intermediate result having unbounded range and precision.
**VSX Scalar Negative Multiply-Add Quad-Precision [using round to Odd] X-form**

\[
xsnmaddqp \text{ VRT, VRA, VRB (} RO=1 \text{)}
\]

Let \( src1 \) be the floating-point value in \( VSR[VRA+32] \) represented in quad-precision format.

Let \( src2 \) be the floating-point value in \( VSR[VRT+32] \) represented in quad-precision format.

Let \( src3 \) be the floating-point value in \( VSR[VRB+32] \) represented in quad-precision format.

If either \( src1 \), \( src2 \), or \( src3 \) is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If \( src1 \) is an Infinity value and \( src3 \) is a Zero value, or if \( src1 \) is a Zero value and \( src3 \) is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If \( src2 \) and the product of \( src1 \) and \( src3 \) are Infinity values having opposite signs, an Invalid Operation exception occurs and VXISI is set to 1.

If \( src1 \) is a Signalling NaN, the result is the Quiet NaN corresponding to \( src1 \).

Otherwise, if \( src1 \) is a Quiet NaN, the result is \( src1 \).

Otherwise, if \( src2 \) is a Signalling NaN, the result is the Quiet NaN corresponding to \( src2 \).

Otherwise, if \( src2 \) is a Quiet NaN, the result is \( src2 \).

Otherwise, if \( src3 \) is a Signalling NaN, the result is the Quiet NaN corresponding to \( src3 \).

Otherwise, if \( src3 \) is a Quiet NaN, the result is \( src3 \).

Otherwise, if \( src1 \) is an Infinity value and \( src3 \) is a Zero value, or if \( src1 \) is a Zero value and \( src3 \) is an Infinity value, the result is the default Quiet NaN.

Otherwise, if the product of \( src1 \) and \( src3 \), and \( src2 \) are Infinity values having opposite signs, the result is the default Quiet NaN.

Otherwise, do the following.

\( src1 \) is multiplied by \( src3 \), producing a product having unbounded significand precision and exponent range.

See part 1 of Table 83. "Actions for xsmadd(a|m)dp".

\( src2 \) is added to the product, producing a sum having unbounded range and precision.

See part 2 of Table 83. "Actions for xsmadd(a|m)dp".

If the intermediate result is Tiny (i.e., the unbiased exponent is less than \( -16382 \)) and \( UE=0 \), the significand is shifted right \( N \) bits, where \( N \) is the difference between \( -16382 \) and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value \( -16382 \).

If \( RO=1 \), let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.


The result is negated and placed into \( VSR[VRT+32] \) in quad-precision format.

FPFR is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FL is set to indicate the result is inexact.

---

1. The quad-precision default Quiet NaN is the value, \( 0x7FFF_8000_0000_0000_0000_0000_0000_0000 \).
If a trap-disabled Invalid Operation exception occurs, \( FR \) and \( FI \) are set to 0.

If a trap-enabled Invalid Operation exception occurs, \( VSR[VRT+32] \) and \( FPRF \) are not modified, and \( FR \) and \( FI \) are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

**Special Registers Altered:**

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX</td>
<td>VXSNAN</td>
<td>VXIMZ</td>
</tr>
</tbody>
</table>

**VSR Data Layout for xsnmaddqp[o]**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>( VSR[VRT+32] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>( VSR[VRT+32] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>( VSR[VRT+32] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>( VSR[VRT+32] )</td>
</tr>
</tbody>
</table>
**Chapter 7. Vector-Scalar Extension Facility**

### Part 1: Multiply

<table>
<thead>
<tr>
<th>src3</th>
<th>src1</th>
<th>p</th>
<th>vximz_flag</th>
<th>vxsnan_flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>-Infinity</td>
<td>p ← +Infinity</td>
<td>vximz_flag ← 1</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Infinity</td>
<td>-NZF</td>
<td>p ← +NZF</td>
<td>vximz_flag ← 1</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Infinity</td>
<td>-Zero</td>
<td>p ← +Zero</td>
<td>vximz_flag ← 1</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Infinity</td>
<td>+Zero</td>
<td>p ← +NZF</td>
<td>vximz_flag ← 1</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Infinity</td>
<td>+NZF</td>
<td>p ← +Infinity</td>
<td>vximz_flag ← 1</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Infinity</td>
<td>+Infinity</td>
<td>p ← +Infinity</td>
<td>vximz_flag ← 1</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Infinity</td>
<td>QNaN</td>
<td>p ← src1</td>
<td>vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>-Infinity</td>
<td>SNaN</td>
<td>p ← +src1</td>
<td>vxsnan_flag ← 1</td>
<td></td>
</tr>
</tbody>
</table>

### Part 2: Add

<table>
<thead>
<tr>
<th>src2</th>
<th>src1</th>
<th>v</th>
<th>vxisi_flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>vxisi_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>-NZF</td>
<td>v ← Add(p, src2)</td>
<td>vxisi_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>vxisi_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Zero</td>
<td>v ← +Zero</td>
<td>vxisi_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>+NZF</td>
<td>v ← +NZF</td>
<td>vxisi_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>vxisi_flag ← 1</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← src2</td>
<td>vxisi_flag ← 1</td>
</tr>
</tbody>
</table>

Explanation:
- **src1**: The quad-precision floating-point value in VSR[VRA+32].
- **src2**: The quad-precision floating-point value in VSR[VRT+32].
- **src3**: The quad-precision floating-point value in VSR[VRB+32].
- **dQNaN**: Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **quiet(x)**: Return a QNaN with the payload of x.
- **Add(x, y)**: Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
  - Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- **Mul(x, y)**: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p**: The intermediate product having unbounded range and precision.
- **v**: The intermediate result having unbounded range and precision.

Table 103. Actions for xsnmaddqp[o]
VSX Scalar Negative Multiply-Subtract Type-A Double-Precision XX3-form

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>A</th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

xsnmsubadp XT,XA,XB

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[32×AX + A].

For xsnmsubadp, do the following.
– Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[32×TX + T].
– Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[32×BX + B].

For xsnmsubmdp, do the following.
– Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[32×BX + B].
– Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[32×TX + T].

src1 is multiplied by src3, producing a product having unbounded range and precision.

See part 1 of Table 104.

src2 is negated and added to the product, producing a sum having unbounded range and precision.

The sum is normalized.

See part 2 of Table 104.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 101, “Scalar Floating-Point Final Result with Negation,” on page 768.

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

Power ISA™ I
Special Registers Altered

| FPRF | FR | FI |
| FX | OX | UX | XX | VXSNAN | VXSI | VXIMZ |

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsnmsubadp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XT].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

VSR Data Layout for xsnmsubmdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XT].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>
**Table 104. Actions for xsnmsub(a|m)dp**

<table>
<thead>
<tr>
<th>src3</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← –Infinity</td>
<td>p ← –Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vximz_flag ← 1</td>
</tr>
<tr>
<td>–NZF</td>
<td>p ← +Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← M(src1,src3)</td>
<td>p ← –Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vximz_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← –Zero</td>
<td>p ← –Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← src3</td>
<td>p ← src3</td>
<td>p ← Q(src3) vximz_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← +Infinity</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

**Part 2: Subtract**

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>v ← dQNaN vxixi_flag ← 1</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
</tr>
<tr>
<td>–NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(p,src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(p,src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
</tbody>
</table>

**Explanation:**

**src1** The double-precision floating-point value in doubleword element 0 of VSR[XA].

**src2** For xsnmsubadp, the double-precision floating-point value in doubleword element 0 of VSR[XB].

**src3** For xsnmsubadp, the double-precision floating-point value in doubleword element 0 of VSR[XC].

**dQNaN** Default quiet NaN (0x7FF8_0000_0000_0000).

**NZF** Nonzero finite number.

**Rezd** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.

**Q(x)** Return a QNaN with the payload of x.

**S(x,y)** Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).

**M(x,y)** Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.

**p** The intermediate product having unbounded range and precision.

**v** The intermediate result having unbounded range and precision.
VSX Scalar Negative Multiply-Subtract Type-A
Single-Precision XX3-form

```
xsnmsubasp  XT,XA,XB

0  6  11  16  21  145
   60  T  A  B
```

VSX Scalar Negative Multiply-Subtract Type-M
Single-Precision XX3-form

```
xsnmsubmsp  XT,XA,XB

0  6  11  16  21  153
   60  T  A  B
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For `xsnmsubasp`, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For `xsnmsubmsp`, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied\(^1\) by src3, producing a product having unbounded range and precision.

See part 1 of Table 105, “Actions for xsnmsub(a|m)sp,” on page 781.

src2 is negated and added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 105, “Actions for xsnmsub(a|m)sp,” on page 781.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.


The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
See Table 101, “Scalar Floating-Point Final Result with Negation,” on page 768.

**Special Registers Altered**

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX</td>
<td>UX</td>
<td>XX</td>
</tr>
<tr>
<td>VXSNAN</td>
<td>VXI S1</td>
<td>VXIMZ</td>
</tr>
</tbody>
</table>

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

**VSR Data Layout for xsnmsubasp**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[XA].dword[0]</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[0]</td>
</tr>
<tr>
<td>unused</td>
<td>unused</td>
<td>unused</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

**VSR Data Layout for xsnmsubmsp**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[XA].dword[0]</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XB].dword[0]</td>
</tr>
<tr>
<td>unused</td>
<td>unused</td>
<td>unused</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>
Table 105. Actions for xsnmsub(a|m)sp

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>NaN</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
</tr>
</tbody>
</table>

Part 1: Multiply

Explanation:

- src1: The double-precision floating-point value in VSR[XA].dword[0].
- src2: The double-precision floating-point value in VSR[XT].dword[0].
- src3: The double-precision floating-point value in VSR[XB].dword[0].
- dQNaN: Default quiet NaN (0x7FF8_0000_0000_0000).
- NZF: Nonzero finite number.
- Rezd: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- Q(x): Return a QNaN with the payload of x.
- S(x,y): Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
- M(x,y): Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- p: The intermediate product having unbounded range and precision.
- v: The intermediate result having unbounded range and precision.

Chapter 7. Vector-Scalar Extension Facility 781
VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd] X-form

\[
x_{\text{snmsubp}} \quad \text{VRT, VRA, VRB} \\
x_{\text{snmsubpq}} \quad \text{VRT, VRA, VRB}
\]

\[
\begin{array}{cccccc}
0 & 63 & 11 & 16 & 21 & 484 \\
\end{array}
\]

If MSK.VSX=0 then VSX_Unavailable();
reset_xflags();

\[
\begin{align*}
src1 & = \text{bfp\_CONVERT\_FROM\_BFP128}(\text{VSR}[\text{VRA}+32]) \\
src2 & = \text{bfp\_CONVERT\_FROM\_BFP128}(\text{VSR}[\text{VRT}+32]) \\
src3 & = \text{bfp\_CONVERT\_FROM\_BFP128}(\text{VSR}[\text{VRB}+32]) \\
v & = \text{bfp\_MULTIPLY\_ADD}(src1, src3, \text{bfp\_NEGATE}(src2)) \\
\text{rnd} & = \text{bfp\_NEGATE}(\text{bfp\_ROUND\_TO\_BFP128}(\text{RO}, \text{FPSCR.RN}, v)) \\
\text{result} & = \text{bfp128\_CONVERT\_FROM\_BFP128}(\text{rnd})
\end{align*}
\]

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

\[
\begin{align*}
\text{vx\_flag} & \leftarrow vxsnan\_flag \mid vximz\_flag \mid vxisi\_flag \\
\text{ex\_flag} & \leftarrow \text{FPSCR.VE} \mid \text{vx\_flag}
\end{align*}
\]

if ex_flag=0 then do

\[
\begin{align*}
\text{VSR}[\text{VRT}+32] & \leftarrow \text{result} \\
\text{FPSCR.FR} & \leftarrow (\text{vx\_flag}=0) \land \text{inc\_flag} \\
\text{FPSCR.FI} & \leftarrow (\text{vx\_flag}=0) \land \text{vx\_flag}
\end{align*}
\]

Let \( src1 \) be the floating-point value in \( \text{VSR}[\text{VRA}+32] \) represented in quad-precision format.

Let \( src2 \) be the floating-point value in \( \text{VSR}[\text{VRT}+32] \) represented in quad-precision format.

Let \( src3 \) be the floating-point value in \( \text{VSR}[\text{VRB}+32] \) represented in quad-precision format.

If either \( src1 \), \( src2 \), or \( src3 \) is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If \( src1 \) is an Infinity value and \( src3 \) is a Zero value, or if \( src1 \) is a Zero value and \( src3 \) is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If \( src2 \) and the product of \( src1 \) and \( src3 \) are Infinity values having same signs, an Invalid Operation exception occurs and VXISI is set to 1.

If \( src1 \) is a Signalling NaN, the result is the Quiet NaN corresponding to \( src1 \).

Otherwise, if \( src1 \) is a Quiet NaN, the result is \( src1 \).

Otherwise, if \( src2 \) is a Signalling NaN, the result is the Quiet NaN corresponding to \( src2 \).

Otherwise, if \( src3 \) is a Quiet NaN, the result is \( src3 \).

Otherwise, if \( src1 \) is an Infinity value and \( src3 \) is a Zero value, or if \( src1 \) is a Zero value and \( src3 \) is an Infinity value, the result is the default Quiet NaN.[1]

Otherwise, if the product of \( src1 \) and \( src3 \), and \( src2 \) are Infinity values having same signs, the result is the default Quiet NaN.

Otherwise, do the following.

\( src1 \) is multiplied by \( src3 \), producing a product having unbounded significand precision and exponent range.

See part 1 of Table 96, "Actions for xsnmsubp[0]".

\( src2 \) is negated and added to the product, producing a sum having unbounded range and precision.

See part 2 of Table 96, "Actions for xsnmsubpq[0]".

If the intermediate result is Tiny (i.e., the unbiased exponent is less than \(-16382\)) and UE=0, the significand is shifted right \( N \) bits, where \( N \) is the difference between \(-16382\) and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value \(-16382\).

If \( RO=1 \), let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.


The result is negated and placed into \( \text{VSR}[\text{VRT}+32] \) in quad-precision format.

FPFR is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

---

[1] The quad-precision default Quiet NaN is the value, \( \text{x}7FFFF_0000_0000_0000_0000_0000_0000 \).
If a trap-enabled Invalid Operation exception occurs, 
\( \text{VSR}[\text{VRT}+32] \) and \( \text{FPRF} \) are not modified, and \( \text{FR} \) and \( \text{FI} \) are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

**Special Registers Altered:**

\[
\begin{align*}
\text{FPRF} & \quad \text{FR} & \quad \text{FI} \\
\text{FX} & \quad \text{VXSNAN} & \quad \text{VXIMZ} & \quad \text{VXISI} & \quad \text{OX} & \quad \text{UX} & \quad \text{XX}
\end{align*}
\]

**VSR Data Layout for \text{xsnmsubqp}[o]**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[VRA+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[VRB+32]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>0</td>
<td>127</td>
</tr>
</tbody>
</table>
### Part 1: Multiply

<table>
<thead>
<tr>
<th>src3</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + 0Zero</td>
<td>p + NZF</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + Hi</td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + 0Zero</td>
<td>p + NZF</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + Hi</td>
<td></td>
</tr>
<tr>
<td>–Zero</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + 0Zero</td>
<td>p + NZF</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + Hi</td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + 0Zero</td>
<td>p + NZF</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + Hi</td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + 0Zero</td>
<td>p + NZF</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + Hi</td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + 0Zero</td>
<td>p + NZF</td>
<td>p + Hi</td>
<td>p + QNaN</td>
<td>p + Hi</td>
<td></td>
</tr>
</tbody>
</table>

**QNaN**

- Vxiimz_flag ← 1
- p ← +Hi

**SNaN**

- p ← quiet(src3)
- Vxsnan_flag ← 1

---

### Part 2: Subtract

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>v + QNaN</td>
<td>v + 0Zero</td>
<td>v + NZF</td>
<td>v + Hi</td>
<td>v + QNaN</td>
<td>v + Hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td>v + QNaN</td>
<td>v + 0Zero</td>
<td>v + NZF</td>
<td>v + Hi</td>
<td>v + QNaN</td>
<td>v + Hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–Zero</td>
<td>v + QNaN</td>
<td>v + 0Zero</td>
<td>v + NZF</td>
<td>v + Hi</td>
<td>v + QNaN</td>
<td>v + Hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v + QNaN</td>
<td>v + 0Zero</td>
<td>v + NZF</td>
<td>v + Hi</td>
<td>v + QNaN</td>
<td>v + Hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v + QNaN</td>
<td>v + 0Zero</td>
<td>v + NZF</td>
<td>v + Hi</td>
<td>v + QNaN</td>
<td>v + Hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>v + QNaN</td>
<td>v + 0Zero</td>
<td>v + NZF</td>
<td>v + Hi</td>
<td>v + QNaN</td>
<td>v + Hi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**QNaN & src1 is a NaN**

- v ← p

**QNaN & src1 not a NaN**

- v ← src2
- Vxsnan_flag ← 1

---

**Explanation:**

- **src1**: The quad-precision floating-point value in `VSR[VR+32]`
- **src2**: The quad-precision floating-point value in `VSR[VR+32]`
- **src3**: The quad-precision floating-point value in `VSR[VR+32]`
- **QNaN**: Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000)
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **quiet(x)**: Return a QNaN with the payload of x.
- **sub(x, y)**: Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
  - Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
- **Mul(x, y)**: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- p: The intermediate product having unbounded range and precision.
- v: The intermediate result having unbounded range and precision.

---

**Table 106. Actions for xsnmsubqp[O]**
VSX Scalar Round to Double-Precision Integer using round to Nearest Away XX2-form

`xsrdpi`    `XT,XB`

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>73</th>
<th>B (X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>III</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

src ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])
rnd ← bfp_ROUND_TO_INTEGER_NEAR_AWAY(src)
result ← bfp64_CONVERT_FROM_BFP(rnd)
if vsxnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vsxnan_flag
i
f vex_flag=0 then do
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
```

Let XT be the value $32 \times TX + T$.
Let XB be the value $32 \times BX + B$.

Let `src` be the double-precision floating-point value in doubleword element 0 of `VSR[XB]`.

`src` is rounded to an integer using the rounding mode Round to Nearest Away.

The result is placed into doubleword element 0 of `VSR[XT]` in double-precision format.

The contents of doubleword element 1 of `VSR[XT]` are set to 0.

`FPRF` is set to the class and sign of the result. `FR` is set to 0. `FI` is set to 0.

If a trap-enabled invalid operation exception occurs, `VSR[XT]` and `FPRF` are not modified, and `FR` and `FI` are set to 0.

Special Registers Altered

`FPRF FR=0b0 FI=0b0 VXSNAN`

Programming Note

This instruction can be used to operate on a single-precision source operand.

---

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

---

VSR Data Layout for xsrdpi

```
src
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[XB].dword[0]</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
</tbody>
</table>

tgt
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[XB].dword[0]</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
</tbody>
</table>
```
**VSX Scalar Round to Double-Precision Integer exact using Current rounding mode XX2-form**

Let XT be the value $32 \times \text{T} + \text{T}$.
Let XB be the value $32 \times \text{BX} + \text{B}$.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

### Special Registers Altered

- FPRF
- FR
- FI
- XX
- VXSNAN

**Programming Note**

This instruction can be used to operate on a single-precision source operand.

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

### VSR Data Layout for xsrdpic

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

Let $\text{XT}$ be the value $32 \times \text{T} + \text{T}$.
Let $\text{XB}$ be the value $32 \times \text{BX} + \text{B}$.
**VSX Scalar Round to Double-Precision Integer using round toward -Infinity XX2-form**

**xsrdpim** XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>II</th>
<th>B</th>
<th>121</th>
<th>87</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])

rnd ← bfp_ROUND_TO_INTEGER_FLOOR(src)

result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do

VSR[32×TX+T].dword[0] ← result

VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000

FPSCR.FPRF ← fprf_CLASS_BFP64(result)

end

FPSCR.FR ← 0b0

FPSCR.FI ← 0b0

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round toward -Infinity.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

**Special Registers Altered**

FPRF FR=0b0 FI=0b0 FX VXSNAN

**Programming Note**

This instruction can be used to operate on a single-precision source operand.

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

<table>
<thead>
<tr>
<th>s r c</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>6x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>
VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form

\[
\text{xsrdpip \ XT,XB}
\]

\[
\begin{array}{cccccc}
0 & 60 & T & 11 & B & 105 \\
\end{array}
\]

\[
\text{if MSR.VSX=0 then VSX_Unavailable()}
\]

\[
\text{reset_xflags()}
\]

\[
\text{src} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP64(VSR[XB].dword[0])}
\]

\[
\text{rnd} \leftarrow \text{bfp\_ROUND\_TO\_INTEGER\_CIEL(src)}
\]

\[
\text{result} \leftarrow \text{bfp64\_CONVERT\_FROM\_BFP(rnd)}
\]

\[
\text{if vsxnan\_flag=1 then SetFX(FPSCR.VXSNAN)}
\]

\[
\text{vex\_flag} \leftarrow \text{FPSCR.VE \& vsxnan\_flag}
\]

\[
\text{if vex\_flag=0 then do}
\]

\[
\text{VSR}[32\times T+T].\text{dword}[0] \leftarrow \text{result}
\]

\[
\text{VSR}[32\times T+T].\text{dword}[1] \leftarrow 0x0000_0000_0000_0000
\]

\[
\text{FPSCR.FPRF} \leftarrow \text{fprf\_CLASS\_BFP64(result)}
\]

\[
\text{end}
\]

\[
\text{FPSCR.FR} \leftarrow 0b0
\]

\[
\text{FPSCR.FI} \leftarrow 0b0
\]

\text{Let XT be the value } 32\times T + T. \text{ Let XB be the value } 32\times B + B. \text{ Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].}

\[
\text{src} \text{ is rounded to an integer using the rounding mode Round toward +Infinity.}
\]

\text{The result is placed into doubleword element 0 of VSR[XT] in double-precision format.}

\text{The contents of doubleword element 1 of VSR[XT] are set to 0.}

\text{FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.}

\text{If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.}

\text{Special Registers Altered}

\[
\text{FPRF FR=0b0 FI=0b0 FX VXSNAN}
\]

\text{Programming Note}

This instruction can be used to operate on a single-precision source operand.

\text{Programming Note}

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
VSX Scalar Round to Double-Precision Integer using round toward Zero XX2-form

```
xsrpiz  XT,XB

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

src ← bfp_CONVERT_FROM_BFP64(VSR[XB].dword[0])
rnd ← bfp_ROUNTO_INTEGER_TRUNC(src)
result ← bfp64_CONVERT_FROM_BFP(rnd)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
vex_flag ← FPSCR.VE & vxsnan_flag
if vex_flag=0 then do
  VSR[32×TX+T].dword[0] ← result
  VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
  FPSCR.FPRF ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
```

\[ src \] is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into doubleword element 0 of \( VSR[XT] \) in double-precision format.

The contents of doubleword element 1 of \( VSR[XT] \) are set to 0.

\( FPRF \) is set to the class and sign of the result. \( FR \) is set to 0. \( FI \) is set to 0.

If a trap-enabled invalid operation exception occurs, \( VSR[XT] \) and \( FPRF \) are not modified, and \( FR \) and \( FI \) are set to 0.

**Special Registers Altered**

\[
FPRF \ FR=0b0 \ FI=0b0 \ FX \ VXSNAN
\]

**Programming Note**

This instruction can be used to operate on a single-precision source operand.

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

**VSR Data Layout for xsrpiz**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>6x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

Let \( XT \) be the value \( 32×TX + T \).

Let \( XB \) be the value \( 32×BX + B \).

Let \( src \) be the double-precision floating-point value in doubleword element 0 of \( VSR[XB] \).
**VSX Scalar Reciprocal Estimate Double-Precision XX2-form**

xsredp XT, XB

<table>
<thead>
<tr>
<th>60</th>
<th>56</th>
<th>T</th>
<th>11</th>
<th>B</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>6</td>
<td>21</td>
<td>90</td>
</tr>
</tbody>
</table>

if MSR.VSX = 0 then VSX_Unavailable() |

reset_flags()

\[
\begin{align*}
\text{src} & \leftarrow \text{bfp\_CONVERT\_FROM\_BFP64(VSR[VB+32].dword[0])} \\
\text{v} & \leftarrow \text{bfp\_RECI\_ESTIMATE(\text{src})} \\
\text{rnd} & \leftarrow \text{bfp\_ROUND\_TO\_BFP64(0b0, FPSCR.RN, v)} \\
\text{result} & \leftarrow \text{bfp64\_CONVERT\_FROM\_BFP(rnd)}
\end{align*}
\]

if vxsnan_flag = 1 then SetFX(FPSCR.VXSNAN)
if oz_flag = 1 then SetFX(FPSCR.OX)
if ux_flag = 1 then SetFX(FPSCR.UX)
if zx_flag = 1 then SetFX(FPSCR.ZX)

\[
\begin{align*}
\text{vex flag} & \leftarrow \text{FPSCR.VE \& vxsnan_flag} \\
\text{zex flag} & \leftarrow \text{FPSCR.ZE \& zx_flag}
\end{align*}
\]

if vex_flag = 0 & zex_flag = 0 then do
    VSR[32×TX+T].dword[0] ← result
    VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
    FPSCR.FPRF ← frf\_CLASS\_BFP64(result)
    FPSCR.FR ← 0bU
    FPSCR.FI ← 0bU
end

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A double-precision floating-point estimate of the reciprocal of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of src. That is,

\[
\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}
\]

Operation with various special values of the operand is summarized below.

### VSR Data Layout for xsredp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

#### Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
### VSX Scalar Reciprocal Estimate

**Single-Precision XX2-form**

**xsresp** XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>I</th>
<th>B</th>
<th>26</th>
<th>B(11)</th>
</tr>
</thead>
</table>

- If MSR.VSX=0 then VSX_Unavailable()
- reset_xflags()

```plaintext
src ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])
v ← bfp_RECIROCAL_ESTIMATE(src)
rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64.Convert_FROM_BFP(rnd)
```

- if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
- if ox_flag=1 then SetFX(FPSCR.OX)
- if ux_flag=1 then SetFX(FPSCR.UX)
- if 0bU then SetFX(FPSCR.XX)
- if zx_flag=1 then SetFX(FPSCR.ZX)

```plaintext
vex_flag ← FPSCR.VE & vxsnan_flag
zex_flag ← FPSCR.ZE & zx_flag
```

- if vex_flag=0 & zex_flag=0 then do
- VSR[32×TX+T].dword[0] ← result64
- VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
- FPSCR.FR ← fprf_CLASS_BFP32(result32)
- FPSCR.FI ← 0bU
- end

- else do
- FPSCR.FR ← 0b0
- FPSCR.FI ← 0b0
- end

Let **XT** be the value \(32\times TX + T\).

Let **XB** be the value \(32\times BX + B\).

Let **src** be the double-precision floating-point value in doubleword element 0 of **VSR[XB]**.

A single-precision floating-point estimate of the reciprocal of **src** is placed into doubleword element 0 of **VSR[XT]** in double-precision format.

Unless the reciprocal of **src** would be a zero, an infinity, the result of a trap-disabled Overflow exception, or a QNaN, the estimate has a relative error in precision no greater than one part in \(16384\) of the reciprocal of **src**. That is,

\[
\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}
\]

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>−Infinity</td>
<td>−Zero</td>
<td>None</td>
</tr>
<tr>
<td>−Zero</td>
<td>−Infinity(^1)</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity(^1)</td>
<td>ZX</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN(^2)</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN(^2)</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if ZE=1.
2. No result if VE=1.

The contents of doubleword element 1 of **VSR[XT]** are set to 0.

**FPRF** is set to the class and sign of the result as represented in single-precision format. **FR** is set to an undefined value. **FI** is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, **VSR[XT]** and **FPRF** are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

**Special Registers Altered**

- **FPRF FR=0bU FI=0bU**
- **FX OX UX ZX XX=0bU VXSNAN**

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
### VSR Data Layout for xsresp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[XB].dword[0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unused</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Let R and RMC specify the rounding mode as follows.

<table>
<thead>
<tr>
<th>R</th>
<th>RMC</th>
<th>FPSCR.RN</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>–</td>
<td>Round to Nearest Away</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>–</td>
<td>reserved</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>00-01</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>00-01</td>
<td>Round towards Zero</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>00-01</td>
<td>Round towards +Infinity</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>00-01</td>
<td>Round towards -Infinity</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>00-01</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>00-01</td>
<td>Round towards Zero</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>00-01</td>
<td>Round towards +Infinity</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>00-01</td>
<td>Round towards -Infinity</td>
</tr>
</tbody>
</table>

Let src be the floating-point value in VSR[VRT+32] represented in quad-precision format.

If src is a Signalling NaN, an Invalid Operation exception occurs, VXSNAN is set to 1, and the result is the Quiet NaN corresponding to the Signalling NaN.

Otherwise, if src is a Quiet NaN, an Infinity, or a Zero, then the result is src.

Otherwise, src is rounded to an integer using the rounding mode rmode.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result.

For xsrqi, FR is set to 0, FI is set to 0, and XX is not set by an Inexact exception.

For xsrqpix, FR is set to indicate if the result was incremented when rounded, FI is set to indicate the result is inexact, and XX is set by an Inexact exception.

If a trap-disabled Invalid Operation exception occurs, FPRF is set to an undefined value.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

Special Registers Altered:
FPRF VXSNAN FX
FR (set to 0) FI (set to 0) .................. (if xsrqi) FR FI XX .............................. (if xsrqpix)
### VSR Data Layout for xsrqpi

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

| 0   | 127         |
Let $R$ and $RMC$ specify the rounding mode as follows.

<table>
<thead>
<tr>
<th>$R$</th>
<th>$RMC$</th>
<th>FPSCR.RN</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>Round to Nearest Away</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>reserved</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>reserved</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>Round to Zero</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>Round to +Infinity</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>Round to Zero</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>Round to +Infinity</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>Round to -Infinity</td>
</tr>
</tbody>
</table>

Let $\text{src}$ be the floating-point value in $\text{VSR}[\text{VRB}+32]$ represented in quad-precision format.

If $\text{src}$ is a Signalling NaN, an Invalid Operation exception occurs, $\text{VXSNAN}$ is set to 1, and the result is the Quiet NaN corresponding to the Signalling NaN, with the significand truncated to double-extended-precision.

Otherwise, if $\text{src}$ is a Quiet NaN, then the result is $\text{src}$ with the significand truncated to double-extended-precision.

Otherwise, if $\text{src}$ is an Infinity or a Zero, the result is $\text{src}$.

Otherwise, $\text{src}$ is rounded to double-extended precision (i.e., 15-bit exponent range and 64-bit significand precision) using the specified rounding mode.


The result is placed into $\text{VSR}[\text{VRT}+32]$ in quad-precision format.

$\text{FPRF}$ is set to the class and sign of the result. $\text{FR}$ is set to indicate if the rounded result was incremented. $\text{FI}$ is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, $\text{FPRF}$ is set to an undefined value, and $\text{FR}$ and $\text{FI}$ are set to 0.

If a trap-enabled Invalid Operation exception occurs, $\text{VSR}[\text{VRT}+32]$ and $\text{FPRF}$ are not modified, and $\text{FR}$ and $\text{FI}$ are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.
Special Registers Altered:

| FPRF | FR | FI | FX | VXSNAN | OX | UX | XX |

**VSR Data Layout for xsrqpxp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tgt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VSX Scalar Round to Single-Precision

**XX2-form**

\[ \text{xsrsp} \rightarrow X_T, X_B \]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>60</th>
<th>281</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

\[
\begin{align*}
\text{src} & \leftarrow \text{bfp}_6\text{CONVERT}\_\text{FROM}\_\text{BFP64}(\text{VSR}[32\times X_B+0].\text{dword}[0]) \\
\text{rmd} & \leftarrow \text{bfp}_8\text{ROUND}\_\text{TO}\_\text{BFP32}(\text{FPSCR.RN, src}) \\
\text{result32} & \leftarrow \text{bfp}_3\text{2}\_\text{CONVERT}\_\text{FROM}\_\text{BFP}\_\text{RED}(\text{rmd}) \\
\text{result64} & \leftarrow \text{bfp}_6\text{4}\_\text{CONVERT}\_\text{FROM}\_\text{BFP}(\text{rmd}) \\
\end{align*}
\]

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

if ox_flag=1 then SetFX(FPSCR.OX)

if ux_flag=1 then SetFX(FPSCR.UX)

if xx_flag=1 then SetFX(FPSCR.XX)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do

\[
\begin{align*}
\text{VSR}[32\times X_T+0].\text{dword}[0] & \leftarrow \text{result64} \\
\text{VSR}[32\times X_T+1].\text{dword}[1] & \leftarrow 0x0000_0000_0000_0000 \\
\text{FPSCR.FPRF} & \leftarrow \text{fprf}\_\text{CLASS}\_\text{BFP32}(\text{result32}) \\
\text{FPSCR.FR} & \leftarrow \text{inc_flag} \\
\text{FPSCR.FI} & \leftarrow \text{xx_flag} \\
\end{align*}
\]

end else do

\[
\begin{align*}
\text{FPSCR.FR} & \leftarrow 0b1 \\
\text{FPSCR.FI} & \leftarrow 0b1 \\
\end{align*}
\]

end

Let \( X_T \) be the value \( 32\times X_T + T \).

Let \( X_B \) be the value \( 32\times X_B + B \).

Let \( \text{src} \) be the double-precision floating-point value in doubleword element 0 of \( \text{VSR}[X_B] \).

\( \text{src} \) is rounded to single-precision using the rounding mode specified by \( \text{RN} \).


The result is placed into doubleword element 0 of \( \text{VSR}[X_T] \) in double-precision format.

The contents of doubleword element 1 of \( \text{VSR}[X_T] \) are set to 0.

\( \text{FPRF} \) is set to the class and sign of the result as represented in single-precision format.

If a trap-enabled invalid operation exception occurs, \( \text{VSR}[X_T] \) and \( \text{FPRF} \) are not modified.

**Special Registers Altered**

\[
\begin{align*}
\text{FPRF} & \quad \text{FR} \quad \text{FI} \quad \text{FX} \quad \text{OX} \quad \text{UX} \quad \text{XX} \quad \text{VXSNAN}
\end{align*}
\]

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

**VSR Data Layout for xsrsp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSX[X_B].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSX[X_B].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
</table>
**VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form**

`xsrsqrtedp XT,XB`

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>11</th>
<th>H</th>
<th>25</th>
<th>25</th>
<th>74</th>
<th>B</th>
<th>B</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

`src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])`

`v ← bfp_RECIPROCAL_SQUARE_ROOT_ESTIMATE(src)`

`rnd ← bfp_ROUND_TO_BFP64(0b0, FPSCR.RN, v)`

`result ← bfp64_CONVERT_FROM_BFP(rnd)`

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)

if zx_flag=1 then SetFX(FPSCR.ZX)

vx_flag ← vxsnan_flag | vxsqrt_flag

vex_flag ← FPSCR.VE & vx_flag

zex_flag ← FPSCR.ZE & zx_flag

if vex_flag=0 & zex_flag=0 then do

VSR[32×TX+T].dword[0] ← result

VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000

FPSCR.FR ← fprf_CLASS_BFP64(result)

FPSCR.FR ← 0bU

FPSCR.FI ← 0bU

end

Let `XT` be the value `32×TX + T`.

Let `XB` be the value `32×BX + B`.

Let `src` be the double-precision floating-point value in doubleword element 0 of `VSR[XB]`.

A double-precision floating-point estimate of the reciprocal square root of `src` is placed into doubleword element 0 of `VSR[XT]` in double-precision format.

Unless the reciprocal of the square root of `src` would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of `src`. That is,

\[
\frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}}} \leq \frac{1}{16384}
\]

**VSR Data Layout for xsrsqrtedp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infinity</td>
<td>QNaN1</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>Finite</td>
<td>QNaN1</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>Zero</td>
<td>Infinity2</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity2</td>
<td>ZX</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN1</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if VE=1.
2. No result if ZE=1.

The contents of doubleword element 1 of `VSR[XT]` are set to 0.

FPRF is set to the class and sign of the result. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, `VSR[XT]` and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

**Special Registers Altered**

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR=0bU</th>
<th>FI=0bU</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX</td>
<td>XX=0bU</td>
<td>VXSQRT</td>
</tr>
</tbody>
</table>

**Programming Note**

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
VSX Scalar Reciprocal Square Root Estimate
Single-Precision XX2-form

\[ \text{estimate} = \frac{1}{\sqrt{r_c}} \leq \frac{1}{16384} \]

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>-Finite</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>-Zero</td>
<td>-Infinity(^2)</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity(^2)</td>
<td>ZX</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if VE=1.
2. No result if ZE=1.

The contents of doubleword element 1 of VSR\([XT]\) are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR\([XT]\) and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FPRF FR=0bU FI=0bU
FX OX UX ZK XX=0bU VXSNAN VXSQRT

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

Let \(XT\) be the value \(32\times TX + T\).

Let \(XB\) be the value \(32\times BX + B\).

Let src be the double-precision floating-point value in doubleword element 0 of VSR\([XB]\).

A single-precision floating-point estimate of the reciprocal square root of src is placed into doubleword element 0 of VSR\([XT]\) in double-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,
## VSR Data Layout for xsrsqrtesp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>

Version 3.1
VSX Scalar Square Root Double-Precision

**XX2-form**

`xssqrtdp XT, XB`

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>III</th>
<th>B</th>
<th>75</th>
<th>B[15:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

v ← bfp_SQUARE_ROOT(src)

rnd ← bfp_ROUND_TO_BFP64(0b0, FPSCR.RN, v)

result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)

if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vxsqrt_flag

vex_flag ← FPSCR.VE & vx_flag

if vex_flag=8 then do

VSR[32×TX+T].dword[0] ← result

VSR[32×TX+T].dword[1] ← 0b0000_0000_0000_0000_0000_0000_0000_0000

FPSCR.FPRF ← fprf_CLASS_BFP64(result)

FPSCR.FR ← inc_flag

FPSCR.FI ← xx_flag

end
else do

FPSCR.FR ← 0b1

FPSCR.FI ← 0b1
end

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Let the unbounded-precision square root of src be produced.

See Table 107.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.


The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

Special Registers Altered

FPRF FR FI FX XX VXSNAN VXSQRT

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
### Table 107. Actions for `xssqrtdp`

<table>
<thead>
<tr>
<th>src</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v ← dQNaN</code></td>
<td><code>v ← dQNaN</code></td>
<td><code>v ← +Zero</code></td>
<td><code>v ← +Zero</code></td>
<td><code>v ← SQRT(src)</code></td>
<td><code>v ← +Infinity</code></td>
<td><code>v ← src</code></td>
<td><code>v ← Q(src)</code></td>
<td><code>vssnan_flag ← 1</code></td>
</tr>
</tbody>
</table>

**Explanation:**

- **src**: The double-precision floating-point value in doubleword element 0 of `VSR[XB]`.
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **SQRT(x)**: The unbounded-precision square root of the floating-point value `x`.
- **Q(x)**: Return a QNaN with the payload of `x`.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Scalar Square Root Quad-Precision [using round to Odd] X-form

\[ \text{vsqrtqp} \rightarrow \text{VRT,VRB} \quad (R0=0) \]
\[ \text{vsqrtqpo} \rightarrow \text{VRT,VRB} \quad (R0=1) \]

<table>
<thead>
<tr>
<th>63</th>
<th>6</th>
<th>27</th>
<th>21</th>
<th>804</th>
<th>RO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>27</td>
<td>21</td>
<td>804</td>
<td>0</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

\[ \text{src} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP128(VSR[VRB+32])} \]
\[ \text{v} \leftarrow \text{bfp\_SQUARE\_ROOT(src)} \]
\[ \text{rnd} \leftarrow \text{bfp\_ROUND\_TO\_BFP128(RO,FPSCR.RN,v)} \]
\[ \text{result} \leftarrow \text{bfp128\_CONVERT\_FROM\_BFP(rnd)} \]

if vsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
if xx_flag=1 then SetFX(FPSCR.XX)

\[ \text{vx\_flag} \leftarrow \text{vxnan\_flag | vxsqrt\_flag} \]
\[ \text{vex\_flag} \leftarrow \text{FPSCR.VE \& vx\_flag} \]

if vx\_flag=0 then
\[ \text{VSR[VRT+32]} \leftarrow \text{result} \]
\[ \text{FPSCR.FPRF} \leftarrow \text{fprf\_CLASS\_BFP128(result)} \]
end

\[ \text{FPSCR.FR} \leftarrow (\text{vx\_flag=0}) \& \text{inc\_flag} \]
\[ \text{FPSCR.FI} \leftarrow (\text{vx\_flag=0}) \& \text{xx\_flag} \]

Let src be the floating-point value in VSR[VRT+32] represented in quad-precision format.

If src is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src is a negative, non-zero value, an Invalid Operation exception occurs and VXSQRT is set to 1.

If src is a Signalling NaN, the result is the Quiet NaN corresponding to src.

Otherwise, if src is a Quiet NaN, the result is src.

Otherwise, if src is a negative value, the result is the default Quiet NaN[1].

Otherwise, do the following.

The normalized square root of src is produced with unbounded significand precision and exponent range.

See Table 108, “Actions for vsqrtqp[0],” on page 804.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.

See Section 7.3.2.6, “Rounding” on page 518 for a description of rounding modes.

If there is loss of precision, an Inexact exception occurs.


The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FPRF is set to an undefined value, and FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

Special Registers Altered:
FPRF FR FI FX VXSNAN VXSQRT XX

VSR Data Layout for vsqrtqp[0]

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>

1. The quad-precision default Quiet NaN is the value, \(0x7FFF_8000_0000_0000_0000_0000_0000_0000\).
### Explanation:

**src**
The quad-precision floating-point value in VSR\[VRB + 32].

**dQNaN**
Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000_0000).

**NZF**
Nonzero finite number.

**sqrt(x)**
Return the normalized\(^1\) square root of floating-point value \( x \), having unbounded significand precision and exponent range.

**quiet(x)**
Convert \( x \) to the corresponding Quiet NaN.

\( v \)
The intermediate result having unbounded significand precision and unbounded exponent range.

---

#### Table 108. Actions for xssqrtqp[o]

1. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

<table>
<thead>
<tr>
<th>src</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← dZero</td>
<td>v ← dZero</td>
<td>v ← sqrt(src)</td>
<td>v ← +Infinity</td>
<td>v ← src</td>
<td>v ← quiet(src)</td>
<td>vxsqrt_flag ← 1</td>
</tr>
</tbody>
</table>

---

\(^1\) Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSX Scalar Square Root Single-Precision $XX^2$-form

The unbounded-precision square root of $src$ is produced.

#### See Table 107.

The intermediate result is rounded to single-precision using the rounding mode specified by $RN$.


The result is placed into doubleword element 0 of $VSR[XT]$ in double-precision format.

The contents of doubleword element 1 of $VSR[XT]$ are set to 0.

$FPFR$ is set to the class and sign of the result as represented in single-precision format. $FR$ is set to indicate if the result was incremented when rounded. $FI$ is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, $VSR[XT]$ and $FPFR$ are not modified, and $FR$ and $FI$ are set to 0.

#### See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

#### Special Registers Altered

$FPFR$ $FR$ $FI$

$FX$ $OX$ $UX$ $XX$ $VXSNAN$ $VXSQRT$

#### Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
### Table 109. Actions for `vssqrtsp`

<table>
<thead>
<tr>
<th>src</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← SQRT(src)</td>
<td>v ← +Infinity</td>
<td>v ← src</td>
<td>v ← Q(src)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>vxsqrt_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

- **src**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **SQRT(x)**: The unbounded-precision and exponent range square root of the floating-point value x.
- **Q(x)**: Return a QNaN with the payload of x.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Scalar Subtract Double-Precision
XX3-form

\( \text{xssubdp} \) \quad XT,XA XB

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| src1 | ← | bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0]) |
| src2 | ← | bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0]) |
| v    | ← | bfp_ADD(src1, bfp_NEGATE(src2)) |
| rnd  | ← | bfp_ROUND_TO_BFP64(0b0, FPSCR.RN, v) |
| result | ← | bfp64_CONVERT_FROM_BFP(rnd) |

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN) if vxisi_flag=1 then SetFX(FPSCR.VXISI) if ox_flag=1 then SetFX(FPSCR.OX) if ux_flag=1 then SetFX(FPSCR.UX)

vx_flag ← vxsnan_flag \| vxisi_flag vex_flag ← FPSCR.VE \& vx_flag

if vex_flag=0 then do
| VSR[32×TX+T].dword[0] | ← | result |
| VSR[32×TX+T].dword[1] | ← | 0x0000_0000_0000_0000_0000_0000_0000_0000 |
| FPSCR.FR | ← | fprf_CLASS_BFP64(result) |
| FPSCR.FR | ← | inc_flag |
| FPSCR.FI | ← | xx_flag |
end else do
| FPSCR.FR | ← | 0b1 |
| FPSCR.FI | ← | 0b1 |
end

Let \( XT \) be the value \( 32×TX + T \).
Let \( XA \) be the value \( 32×AX + A \).
Let \( XB \) be the value \( 32×BX + B \).

Let \( src1 \) be the double-precision floating-point value in doubleword element 0 of VSR[AX].

\( src2 \) is negated and added to \( src1 \), producing a sum having unbounded range and precision.

See Table 110.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.


The result is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, "VSX Scalar Floating-Point Final Result," on page 662.

Special Registers Altered

\( \text{FPRF} \) \( \text{FR} \) \( \text{FI} \) \( \text{FX} \) \( \text{OX} \) \( \text{UX} \) \( \text{XX} \) \( \text{VXSNAN} \) \( \text{VXISI} \)

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xssubdp

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td>VSR[AX].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XB].dword[0]</td>
<td>0x0000_0000_0000_0000_0000_0000</td>
</tr>
</tbody>
</table>

---

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 110. Actions for xssubdp

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>dQNaN</td>
<td>v ← -dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

### Explanation:
- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- **S(x,y)**: The floating-point value y is negated and then added to the floating-point value x.
- **S(x,y)**: Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
- **Note**: If x = y, v is considered to be an exact-zero-difference result (Rezd).
- **Q(x)**: Return a QNaN with the payload of x.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Scalar Subtract Quad-Precision [using round to Odd] X-form

\[ \text{xssubqp: } VRT, VRA, VRB \quad (RO=0) \]
\[ \text{xssubqpo: } VRT, VRA, VRB \quad (RO=1) \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>516</th>
<th>RO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>516</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v ← bfp_ADD(src1, bfp_NEGATE(src2))
rnd ← bfp_ROUND_TO_BFP128(RD, FPSCR.RN, v)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
VSR[VRT+32] ← result
FPSCR.FPRF ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 and src2 are Infinity values having same signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src1 and src2 are Infinity values having same signs, the result is the default Quiet NaN[1].

Otherwise, do the following.

The normalized sum of the negation of src2 added to src1 is produced with unbounded significand precision and exponent range.

See Table 111, “Actions for xssubq[0],” on page 810.

If the intermediate result is Tiny (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right \(N\) bits, where \(N\) is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.


The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FPRF is set to an undefined value, and FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 63, “VSX Scalar Floating-Point Final Result,” on page 662.

Special Registers Altered:

FPRF FR FI FX VXSNAN VXISI OX UX XX

---

1. The quad-precision default Quiet NaN is the value, \(0x7FFFF_0000_0000_0000_0000_0000_0000_0000\).
## VSR Data Layout for xssubqp[o]

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>tgt</th>
</tr>
</thead>
</table>

### Table 111. Actions for xssubqp[o]

<table>
<thead>
<tr>
<th>src1</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v + 0Nan</td>
<td>v = sub(src1,src2)</td>
<td>v = src1</td>
<td>v = sub(src1,src2)</td>
<td>v = src2</td>
<td>v = sub(src1,src2)</td>
<td>v = QNaN</td>
<td>v = src1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v = sub(src1,src2)</td>
<td>v = src2</td>
<td>v = sub(src1,src2)</td>
<td>v = src2</td>
<td>v = sub(src1,src2)</td>
<td>v = QNaN</td>
<td>v = src1</td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>v = src2</td>
<td>v = 0Rez</td>
<td>v = -Zero</td>
<td>v = src2</td>
<td>v = sub(src1,src2)</td>
<td>v = QNaN</td>
<td>v = src1</td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v = sub(src1,src2)</td>
<td>v = 0Rez</td>
<td>v = +Zero</td>
<td>v = sub(src1,src2)</td>
<td>v = src2</td>
<td>v = sub(src1,src2)</td>
<td>v = QNaN</td>
<td>v = src1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v + Infinity</td>
<td>v = src2</td>
<td>v = sub(src1,src2)</td>
<td>v = QNaN</td>
<td>v = src1</td>
<td>v = sub(src1,src2)</td>
<td>v = QNaN</td>
<td>v = src1</td>
</tr>
<tr>
<td>QNaN</td>
<td>v = src1</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
</tr>
<tr>
<td>SNaN</td>
<td>v = src1</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
<td>v = quiet(src1)</td>
</tr>
</tbody>
</table>

**Explanation:**
- **src1**: The quad-precision floating-point value in VSR[VRA+32].
- **src2**: The quad-precision floating-point value in VSR[VRB+32].
- **QNaN**: Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **Rez**: Exact-zero-difference result (subtraction of two finite numbers having same magnitude and signs).
- **sub(x,y)**: Return the normalized difference of floating-point value x and floating-point value y, having unbounded significand precision and exponent range.
- **Note**: If x = y, v is considered to be an exact-zero-difference result (Rez).
- **quiet(x)**: Convert x to the corresponding Quiet NaN.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.
**VSX Scalar Subtract Single-Precision **

**XX3-form**

**xssubsp**  
**XT, XA, XB**

### Code Snippet

```lang
if MSR.VSX=0 then VSX_Unavailable()
reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v ← bfp.ADD(src1, bfp.NEGATE(src2))
result64 ← bfp64_CONVERT_FROM_BFP(v)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxisn_flag=1 then SetFX(FPSCR.VXSI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vxisn_flag
vex_flag ← FPSCR.VE & vx_flag
if vex_flag=0 then do
  VSR[32×TX+T].dword[0] ← result64
  VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
  FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
  FPSCR.FR ← inc_flag
  FPSCR.FI ← xx_flag
else do
  FPSCR.FR ← 0b0
  FPSCR.FI ← 0b0
end
```

### Explanation

Let $src2$ be the double-precision floating-point value in doubleword element 0 of $VSR[XB]$.

$src2$ is negated and added$^{[1]}$ to $src1$, producing the sum, $v$, having unbounded range and precision.

See Table 112, "Actions for xssubsp," on page 812. $v$ is normalized$^{[2]}$ and rounded to single-precision using the rounding mode specified by $RN$.


The result is placed into doubleword element 0 of $VSR[XT]$.

The contents of doubleword element 1 of $VSR[XT]$ are set to 0.

$FPRF$ is set to the class and sign of the result as represented in single-precision format. $FR$ is set to indicate if the result was incremented when rounded. $FI$ is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, $VSR[XT]$ and $FPRF$ are not modified, and $FR$ and $FI$ are set to 0.

See Table 63, "VSX Scalar Floating-Point Final Result," on page 662.

### Special Registers Altered

- $FPFR$  
- $FR$  
- $FI$  
- $FX$  
- $OX$  
- $UX$  
- $XX$  
- $VXSNAN$  
- $VXSI$

### Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

### VSR Data Layout for xssubsp

- **src1**  
  - $VSR[XA].dword[0]$  
  - unused
- **src2**  
  - $VSR[XB].dword[0]$  
  - unused
- **tgt**  
  - $VSR[XB].dword[0]$  
  - $0x0000_0000_0000_0000$

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Explanation:

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- **S(x,y)**: The floating-point value y is negated and then added to the floating-point value x.
- **S(x,y)**: Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
  - **Note**: If x = y, v is considered to be an exact-zero-difference result (Rezd).
- **Q(x)**: Return a QNaN with the payload of x.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.

### Table 112. Actions for xssubsp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← v ← Q(src2)</td>
<td>v ← v ← nan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1,src2)</td>
<td>v ← src1</td>
<td>v ← S(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← Rezd</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← Rezd</td>
<td>v ← +Zero</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1,src2)</td>
<td>v ← src1</td>
<td>v ← S(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

**Note**: The vxisi_flag and vxsnan_flag values are not shown in the table for brevity.
VSX Scalar Test for software Divide Double-Precision XX3-form

**xstdivdp**

<table>
<thead>
<tr>
<th>BF</th>
<th>AX</th>
<th>XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

If MSR.VSX=0 then VSX_Unavailable();

\[
\begin{align*}
\text{src1} & \leftarrow VSR[32\times AX + A].dword[0] \\
\text{src2} & \leftarrow VSR[32\times BX + B].dword[0] \\
e_a & \leftarrow \text{src1}.bit[1:11] - 1023 \\
e_b & \leftarrow \text{src2}.bit[1:11] - 1023 \\
\text{fe_flag} & \leftarrow \text{IsNaN} (\text{src1}) \mid \text{IsInf} (\text{src1}) \mid \text{IsNaN} (\text{src2}) \mid \text{IsInf} (\text{src2}) \mid \text{IsZero} (\text{src2}) \mid (\text{e_b} \leq -1022) \mid (\text{e_b} \geq 1021) \mid (\text{IsZero} (\text{src1}) \& (\text{e_a} - \text{e_b}) \geq 1023) \mid (\text{IsZero} (\text{src1}) \& (\text{e_a} - \text{e_b}) \leq -1021) \mid (\text{IsZero} (\text{src1}) \& \text{e_a} \leq -970)
\end{align*}
\]

\[
\begin{align*}
\text{fg_flag} & \leftarrow \text{IsInf} (\text{src1}) \mid \text{IsInf} (\text{src2}) \mid \text{IsZero} (\text{src2}) \mid \text{IsDen} (\text{src2}) \\
\text{fl_flag} & \leftarrow \text{xsrredp_error}() \ll 14 \\
\text{CR}[BF] & \leftarrow 0b1 || \text{fg_flag} || \text{fe_flag} || 0b0
\end{align*}
\]

Let \( \text{XA} \) be the value \( 32 \times AX + A \).
Let \( \text{XB} \) be the value \( 32 \times BX + B \).

Let \( \text{src1} \) be the double-precision floating-point value in doubleword element 0 of VSR[\( \text{XA} \)].

Let \( \text{src2} \) be the double-precision floating-point value in doubleword element 0 of VSR[\( \text{XB} \)].

Let \( e_a \) be the unbiased exponent of \( \text{src1} \).
Let \( e_b \) be the unbiased exponent of \( \text{src2} \).

\( \text{fe_flag} \) is set to 1 for any of the following conditions.
- \( \text{src1} \) is a NaN or an infinity.
- \( \text{src2} \) is a zero, a NaN, or an infinity.
- \( e_b \) is less than or equal to -1022.
- \( e_b \) is greater than or equal to 1021.
- \( \text{src1} \) is not a zero and the difference, \( e_a - e_b \), is greater than or equal to 1023.
- \( \text{src1} \) is not a zero and the difference, \( e_a - e_b \), is less than or equal to -1021.
- \( \text{src1} \) is not a zero and \( e_a \) is less than or equal to -970.

Otherwise \( \text{fe_flag} \) is set to 0.

\( \text{fg_flag} \) is set to 1 for any of the following conditions.
- \( \text{src1} \) is an infinity.
- \( \text{src2} \) is a zero, an infinity, or a denormalized value.

Otherwise \( \text{fg_flag} \) is set to 0.

\( \text{CR} \) field BF is set to the value 0b1 || \( \text{fg_flag} || \text{fe_flag} || 0b0 \).

**Special Registers Altered**

CR field BF
VSX Scalar Test for software Square Root
Double-Precision XX2-form

xstsqrtdp BF,XB

| 60 | BF | 60  | 106 | B | 21 |

if MSR.VSX=0 then VSX_Unavailable()

src ← VSR[32xBX+B].dword[0]
fe_flag ← IsNaN(src) | IsInf(src) | IsZero(src) | IsNeg(src) | ( e_b <= -970 )
fq_flag ← IsInf(src) | IsZero(src) | IsDen(src)
fl_flag ← xsrsqrtedp_error() <= 2^14

CR.field[BF] ← 0b1 || fq_flag || fe_flag || 0b0

Let XB be the value 32xBX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Let e_b be the unbiased exponent of src.

fe_flag is set to 1 for any of the following conditions.
- src is a zero, a NaN, an infinity, or a negative value.
- e_b is less than or equal to -970

Otherwise fe_flag is set to 0.

fq_flag is set to 1 for any of the following conditions.
- src is a zero, an infinity, or a denormalized value.

Otherwise fq_flag is set to 0.

CR field BF is set to the value 0b1 || fq_flag || fe_flag || 0b0.

Special Registers Altered
CR field BF

VSR Data Layout for xstsqrtdp

<table>
<thead>
<tr>
<th>src2</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>64</td>
</tr>
</tbody>
</table>
**Chapter 7. Vector-Scalar Extension Facility**

### VSX Scalar Test Data Class Double-Precision XX2-form

#### xststdcdp

<table>
<thead>
<tr>
<th>BF</th>
<th>XB</th>
<th>DCMX</th>
<th>B</th>
<th>362</th>
<th>BF/لة</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>362/ب</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable();

src ← VSR[32×BX + B].dword[0]

exponent ← src.bit[3:12]

fraction ← src.bit[12:63]

class.Infinity ← (exponent = 0x7FF) & (fraction = 0)
class.NaN ← (exponent = 0x7FF) & (fraction != 0)
class.Zero ← (exponent = 0x000) & (fraction = 0)
class.Denormal ← (exponent = 0x000) & (fraction != 0)


CR.bit[4×BF+32] ← FPSCR.FL ← src.sign

CR.bit[4×BF+33] ← FPSCR.FG ← 0b0

CR.bit[4×BF+34] ← FPSCR.FE ← match

CR.bit[4×BF+35] ← FPSCR.FU ← 0b0

Let XB be the sum 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Bit 0 of CR field BF and bit 0 of FPCC are set to the sign bit of src.

Bit 1 of CR field BF and bit 1 of FPCC are set to 0b0.

Bit 2 of CR field BF and bit 2 of FPCC are set to indicate whether the data class of src, as represented in double-precision format, matches any of the data classes specified by DCMX (Data Class Mask).

#### DCMX bit Data Class

<table>
<thead>
<tr>
<th>DCMX bit</th>
<th>Data Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>+Infinity</td>
</tr>
<tr>
<td>2</td>
<td>-Infinity</td>
</tr>
<tr>
<td>3</td>
<td>+Zero</td>
</tr>
<tr>
<td>4</td>
<td>-Zero</td>
</tr>
<tr>
<td>5</td>
<td>+Denormal</td>
</tr>
<tr>
<td>6</td>
<td>-Denormal</td>
</tr>
</tbody>
</table>

Bit 3 of CR field BF and bit 3 of FPCC are set to 0b0.

**Special Registers Altered:**

CR field BF

FPCC

#### VSR Data Layout for xststdcdp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>64</td>
</tr>
</tbody>
</table>
Let \( \text{src} \) be the quad-precision floating-point value in \( \text{VSR}[\text{VRB}+32] \).

Let the \( \text{DCMX} \) (Data Class Mask) field specify one or more of the 7 possible data classes, where each bit corresponds to a specific data class.

<table>
<thead>
<tr>
<th>DCM bit</th>
<th>Data Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>+Infinity</td>
</tr>
<tr>
<td>2</td>
<td>Infinity</td>
</tr>
<tr>
<td>3</td>
<td>+Zero</td>
</tr>
<tr>
<td>4</td>
<td>-Zero</td>
</tr>
<tr>
<td>5</td>
<td>+Denormal</td>
</tr>
<tr>
<td>6</td>
<td>-Denormal</td>
</tr>
</tbody>
</table>

Bit 0 of \( \text{CR} \) field \( BF \) and bit 0 of \( \text{FPCC} \) are set to the sign of \( \text{src} \).

Bit 1 of \( \text{CR} \) field \( BF \) and bit 1 of \( \text{FPCC} \) are set to 0b0.

Bit 2 of \( \text{CR} \) field \( BF \) and bit 2 of \( \text{FPCC} \) are set to indicate whether the data class of \( \text{src} \), as represented in quad-precision format, matches any of the data classes specified by \( \text{DCM} \).

Bit 3 of \( \text{CR} \) field \( BF \) and bit 3 of \( \text{FPCC} \) are set to 0b0.

**Special Registers Altered:**

- \( \text{CR} \) field \( BF \)
- \( \text{FPCC} \)
Let $\text{XB}$ be the sum $32\times BX + B$.

Let $src$ be the double-precision floating-point value in doubleword element 0 of $\text{VSR}[XB]$.

Bit 0 of CR field BF and bit 0 of FPCC are set to the sign bit of $src$.

Bit 1 of CR field BF and bit 1 of FPCC are set to 0b0.

Bit 2 of CR field BF and bit 2 of FPCC are set to indicate whether the data class of $src$, as represented in single-precision format, matches any of the data classes specified by $\text{DCMX}$ (Data Class Mask).

**DCMX bit Data Class**

<table>
<thead>
<tr>
<th>DCMX bit</th>
<th>Data Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>+Infinity</td>
</tr>
<tr>
<td>2</td>
<td>-Infinity</td>
</tr>
<tr>
<td>3</td>
<td>+Zero</td>
</tr>
<tr>
<td>4</td>
<td>-Zero</td>
</tr>
<tr>
<td>5</td>
<td>+Denormal</td>
</tr>
<tr>
<td>6</td>
<td>-Denormal</td>
</tr>
</tbody>
</table>

Bit 3 of CR field BF and bit 3 of FPCC are set to indicate if $src$ is not representable in single-precision format.

**Special Registers Altered:**

- CR field BF
- FPCC

---

**VSX Scalar Test Data Class Single-Precision XX2-form**

**VSR Data Layout for xststdcdp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB] . dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>127</td>
<td></td>
<td>127</td>
</tr>
</tbody>
</table>
### VSX Scalar Extract Exponent Double-Precision XX2-form

**xsxexpdp**  
**RT,XB**

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>RT</th>
<th>0</th>
<th>B</th>
<th>347</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()  

src ← VSR[32×BX+B].dword[0]  

gpr[RT] ← (src >> 52) & 0x0000_0000_0000_07FF

Let XB be the sum 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The value of the exponent field in src is placed into gpr[RT] in unsigned integer format.

**Special Registers Altered:**  
None

**Programming Note**  
This instruction can be used to operate on a single-precision source operand.

### VSX Scalar Extract Exponent Quad-Precision X-form

**xsxexpdp**  
**VRT,VRB**

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>VRT</th>
<th>2</th>
<th>VRB</th>
<th>804</th>
<th>61</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()  

src ← VSR[VRB+32]  

VSR[VRT+32].dword[0] ← EXTZ64(src.bit[1:15]), 64|

VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000

Let src be the quad-precision floating-point value in VSR[VRB+32].

The contents of the exponent field of src (bits 1:15) are zero-extended and placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

**Special Registers Altered:**  
None

### VSR Data Layout for xsxexpdp

**src**  
VSR[XB].dword[0]  

**tgt**  
GPR[RT]

### VSR Data Layout for xsxexpdp

**src**  
VSR[VRB+32]

**tgt**  
VSR[VRT+32].dword[0]  
0x0000_0000_0000_0000
## VSX Scalar Extract Significand
### Double-Precision XX2-form

**xsxsigdp**

<table>
<thead>
<tr>
<th>60</th>
<th>RT</th>
<th>11</th>
<th>16</th>
<th>B</th>
<th>347</th>
<th>34</th>
</tr>
</thead>
</table>

- if MSR.VSX=0 then VSX_Unavailable()
- exponent ← VSR[32×BX+B].bit[1:11]
- fraction ← EXTZ64(VSR[32×BX+B].bit[12:63])
- if (exponent != 0) & (exponent != 2047) then
  - significand ← fraction | 0x0010_0000_0000_0000
  - else
  - significand ← fraction
- GPR[RT] ← significand

Let **XB** be the sum 32×BX + B.

Let **src** be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The significand of **src** is placed into GPR[RT] in unsigned integer format. If **src** is a normal value, the implicit leading bit is set to 1.

### Special Registers Altered:
- None

### Programming Note
This instruction can be used to operate on a single-precision source operand.

### VSR Data Layout for xsxsigdp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>GPR[RT]</td>
<td></td>
</tr>
</tbody>
</table>

### VSR Data Layout for xsxsigqp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[VRT+32]</td>
</tr>
</tbody>
</table>
VSX Vector Absolute Double-Precision

**XX2-form**

**xvabsdp**  
XT, XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>11</th>
<th>B</th>
<th>473</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();

\[
\text{do } i = 0 \text{ to } 1 \\
\quad \text{src} \leftarrow VSR[32 \times BX + B].dword[i] \\
\quad VSR[32 \times TX + T].dword[i] \leftarrow \text{bfp64\_ABSOLUTE(src)}
\]

end

Let XT be the value \(32 \times TX + T\).
Let XB be the value \(32 \times BX + B\).

For each integer value \(i\) from 0 to 1, do the following.

The contents of doubleword element \(i\) of VSR[XB], with bit 0 set to 0, is placed into doubleword element \(i\) of VSR[XT].

**Special Registers Altered**

None

**VSR Data Layout for xvabsdp**

\[
\begin{array}{|c|c|c|}
\hline
\text{src} & VSR[XB].dword[0] & VSR[XB].dword[1] \\
\hline
\text{tgt} & VSR[XT].dword[0] & VSR[XT].dword[1] \\
\hline
\end{array}
\]

VSX Vector Absolute Single-Precision

**XX2-form**

**xvabssp**  
XT, XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>11</th>
<th>B</th>
<th>409</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();

\[
\text{do } i = 0 \text{ to } 3 \\
\quad \text{src} \leftarrow VSR[32 \times BX + B].word[i] \\
\quad VSR[32 \times TX + T].word[i] \leftarrow \text{bfp32\_ABSOLUTE(src)}
\]

end

Let XT be the value \(32 \times TX + T\).
Let XB be the value \(32 \times BX + B\).

For each integer value \(i\) from 0 to 3, do the following.

The contents of word element \(i\) of VSR[XB], with bit 0 set to 0, is placed into word element \(i\) of VSR[XT].

**Special Registers Altered**

None

**VSR Data Layout for xvabssp**

\[
\begin{array}{|c|c|c|c|c|}
\hline
\hline
\hline
\end{array}
\]
**VSX Vector Add Double-Precision XX3-form**

\[
\text{xvadddp XT,XA,XB}
\]

Let XT be the value \(32 \times TX + T\).
Let XA be the value \(32 \times AX + A\).
Let XB be the value \(32 \times BX + B\).

For each integer value \(i\) from 0 to 1, do the following.

Let \(s_{rc1}\) be the double-precision floating-point operand in doubleword element \(i\) of VSR[XA].

Let \(s_{rc2}\) be the double-precision floating-point operand in doubleword element \(i\) of VSR[XB].

\(s_{rc2}\) is added \(^1\) to \(s_{rc1}\), producing a sum having unbounded range and precision.

The sum is normalized \(^2\).

See Table 113.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.


The result is placed into doubleword element \(i\) of VSR[XT] in double-precision format.

See Table 114, “Vector Floating-Point Final Result,” on page 823.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

\[
\text{FX OX UX XX VXSNAN VXISI}
\]

---

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 113. Actions for xvadddp (element i)

<table>
<thead>
<tr>
<th>src2</th>
<th>src1</th>
<th>src2</th>
<th>src1</th>
<th>src2</th>
<th>src1</th>
<th>src2</th>
<th>src1</th>
<th>src2</th>
<th>src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← -Zero</td>
<td>v ← Rezd</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Rezd</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v← Q(src1)</td>
<td>v← Q(src1)</td>
<td>v← Q(src1)</td>
<td>v← Q(src1)</td>
<td>v← Q(src1)</td>
<td>v← Q(src1)</td>
<td>v← Q(src1)</td>
<td>v← Q(src1)</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The double-precision floating-point value in doubleword element i of VSR[XA] (where i={0,1}).
- **src2** The double-precision floating-point value in doubleword element i of VSR[XB] (where i={0,1}).
- **dQNaN** Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF** Nonzero finite number.
- **Rezd** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- **A(x,y)** Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- **Q(x)** Return a QNaN with the payload of x.
- **v** The intermediate result having unbounded significand precision and unbounded exponent range.
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>Case</th>
<th>FPSCR.VE</th>
<th>FPSCR.OE</th>
<th>FPSCR.UE</th>
<th>FPSCR.XE</th>
<th>vxsnan_flag</th>
<th>vximz_flag</th>
<th>vxisi_flag</th>
<th>vxidi_flag</th>
<th>vxzdz_flag</th>
<th>vxsqrt_flag</th>
<th>zx_flag</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(r), f(x(ZX))</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(r), f(x(VXZDZ))</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>T(r), f(x(VXISI))</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(r), f(x(VXIMZ))</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(r), f(x(VXSNAN))</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>f(x(VXZDZ)), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>f(x(VXISI)), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>f(x(VXIMZ)), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>f(x(VXSNAN)), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>f(x(VXSNAN), f(xVIXM2)), error()</td>
</tr>
<tr>
<td>Normal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>T(r), f(x(xx))</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>no</td>
<td>-</td>
<td>T(r), f(x(xx))</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>T(r), f(x(xx), error())</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>T(r), f(x(xx), error())</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>T(r), f(x(xx), error())</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>T(r), f(x(xx), error())</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>f(x(DX)), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>no</td>
<td>-</td>
<td>f(x(DX), f(x(xx)), error())</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>f(x(DX), f(x(xx)), error())</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>f(x(DX), f(x(xx)), error())</td>
</tr>
</tbody>
</table>

### Explanation:
- The results do not depend on this condition.
- f(x) The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target precision, unbounded exponent range.
- q The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.
- r The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target precision, bounded exponent range.
- The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
- T(x) The value x is placed in element i of VSR[XT] in the target precision format (where i = {0, 1} for results with 64-bit elements, and i = {0, 1, 3, 4} for results with 32-bit elements).

Table 114.Vector Floating-Point Final Result
The results do not depend on this condition.

\[ f_x(x) \]  FPSCR.FX is set to \( 1 \) if FPSCR.x = 0. FPSCR.x is set to 1.

\[ t \]  The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target precision, unbounded exponent range.

\[ r \]  The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target precision, bounded exponent range.

\[ v \]  The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.

\[ \text{error}() \]  The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.

\[ T(x) \]  The value \( x \) is placed in element \( i \) of VSR[XI] in the target precision format (where \( i = \{0,1\} \) for results with 64-bit elements, and \( i = \{0,1,3,4\} \) for results with 32-bit elements).
VSX Vector Add Single-Precision XX3-form

xvaddsp XT,XA,XB

If MSR.VSX=0 then VSX_Unavailable()

\[ \text{ex_flag} \leftarrow 0b0 \]

\[
\text{do } i = 0 \text{ to } 3 \\
\text{reset_xflags()} \\
\text{src1} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP32(VSR[32\times AX + A].word[i])} \\
\text{src2} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP32(VSR[32\times BX + B].word[i])} \\
\text{v} \leftarrow \text{bfp\_ADD(src1, src2)} \\
\text{rnd} \leftarrow \text{bfp\_ROUND\_TO\_BFP32(FPSCR.RN, v)} \\
\text{vresult.word[i]} \leftarrow \text{bfp32\_CONVERT\_FROM\_BFP(rnd)} \\
\text{if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)} \\
\text{if vxisi_flag=1 then SetFX(FPSCR.VXISI)} \\
\text{if ox_flag=1 then SetFX(FPSCR.OX)} \\
\text{if ux_flag=1 then SetFX(FPSCR.UX)} \\
\text{if xx_flag=1 then SetFX(FPSCR.XX)} \\
\text{ex_flag} \leftarrow \text{ex_flag | (FPSCR.VE & vxsnan_flag)} \\
\text{ex_flag} \leftarrow \text{ex_flag | (FPSCR.VE & vxisi_flag)} \\
\text{ex_flag} \leftarrow \text{ex_flag | (FPSCR.OE & ox_flag)} \\
\text{ex_flag} \leftarrow \text{ex_flag | (FPSCR.UE & ux_flag)} \\
\text{ex_flag} \leftarrow \text{ex_flag | (FPSCR.XE & xx_flag)} \\
\text{end} \\
\text{if ex_flag=0 then VSR[32\times TX + T] \leftarrow vresult}
\]

Special Registers Altered

FX OX UX XX VXSNAN VXISI

Let XT be the value \(32\times TX + T\).
Let XA be the value \(32\times AX + A\).
Let XB be the value \(32\times BX + B\).

For each integer value \(i\) from 0 to 3, do the following.

Let \(\text{src1}\) be the single-precision floating-point operand in word element \(i\) of VSR[XA].

Let \(\text{src2}\) be the single-precision floating-point operand in word element \(i\) of VSR[XB].

\(\text{src2}\) is added\(^1\) to \(\text{src1}\), producing a sum having unbounded range and precision.

The sum is normalized\(^2\).

See Table 115.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.


The result is placed into word element \(i\) of VSR[XT] in single-precision format.

See Table 114, “Vector Floating-Point Final Result,” on page 823.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

VSR Data Layout for xvaddsp

|------|----------------|----------------|----------------|----------------|

---

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the sign of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significant left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 115: Actions for xvaddsp (element i)

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>-Infinity</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← Q(src1)</td>
<td>vxsi_flag ← 1</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← -Zero</td>
<td>v ← Rezd</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Rezd</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>v ← Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The single-precision floating-point value in word element i of VSR[XA] (where i = {0,1,2,3}).
- **src2** The single-precision floating-point value in word element i of VSR[XB] (where i = {0,1,2,3}).
- **dQNaN** Default quiet NaN (0x7FC0_0000).
- **NZF** Nonzero finite number.
- **Rezd** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- **A(x,y)** Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
- ** Note:** If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- **Q(x)** Return a QNaN with the payload of x.
- **v** The intermediate result having unbounded significand precision and unbounded exponent range.
### VSX Vector bfloat16 GER (rank-2 update) XX3-form

`xbf16ger2` AT,XA,XB

<table>
<thead>
<tr>
<th>AT</th>
<th>A</th>
<th>B</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

### VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate XX3-form

`xbf16ger2pp` AT,XA,XB

<table>
<thead>
<tr>
<th>AT</th>
<th>A</th>
<th>B</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

### VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate XX3-form

`xbf16ger2pn` AT,XA,XB

<table>
<thead>
<tr>
<th>AT</th>
<th>A</th>
<th>B</th>
<th>178</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

### VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate XX3-form

`xbf16ger2np` AT,XA,XB

<table>
<thead>
<tr>
<th>AT</th>
<th>A</th>
<th>B</th>
<th>114</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

### VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate XX3-form

`xbf16ger2nn` AT,XA,XB

<table>
<thead>
<tr>
<th>AT</th>
<th>A</th>
<th>B</th>
<th>242</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

### Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) MMIRR:XX3-form

`pmxbf16ger2` AT,XA,XB,XMSK,YMSK,PMSK

<table>
<thead>
<tr>
<th>AT</th>
<th>A</th>
<th>B</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

### Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form

`pmxbf16ger2pp` AT,XA,XB,XMSK,YMSK,PMSK

<table>
<thead>
<tr>
<th>AT</th>
<th>A</th>
<th>B</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

### Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form

`pmxbf16ger2pn` AT,XA,XB,XMSK,YMSK,PMSK

<table>
<thead>
<tr>
<th>AT</th>
<th>A</th>
<th>B</th>
<th>178</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

### Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form

`pmxbf16ger2np` AT,XA,XB,XMSK,YMSK,PMSK

<table>
<thead>
<tr>
<th>AT</th>
<th>A</th>
<th>B</th>
<th>114</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

### Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form

`pmxbf16ger2nn` AT,XA,XB,XMSK,YMSK,PMSK

<table>
<thead>
<tr>
<th>AT</th>
<th>A</th>
<th>B</th>
<th>242</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>
if MSR.VSX=0 then VSX_Unavailable();

if "xvb16ger2" | "xvb16ger2pp" | "xvb16ger2pn" | "xvb16ger2np" | "xvb16ger2nn" then do
  PMSK ← 0b01 // enable all rank updates
  XMSK ← 0b1111 // enable all ACC[AT] rows
  YMSK ← 0b1111 // enable all ACC[AT] columns
end

do i = 0 to 3
  do j = 0 to 3
    if XMSK.bit[i]=1 & YMSK.bit[j]=1 then do
      src11 ← (PMSK.bit[0]=0) ? bfp_ZERO : bfp_CONVERT_FROM_BFLOAT16(VSR[32×AX+A].word[i].hword[0])
      src21 ← (PMSK.bit[0]=0) ? bfp_ZERO : bfp_CONVERT_FROM_BFLOAT16(VSR[32×BX+B].word[j].hword[0])
      src12 ← (PMSK.bit[1]=0) ? bfp_ZERO : bfp_CONVERT_FROM_BFLOAT16(VSR[32×AX+A].word[i].hword[1])
      src22 ← (PMSK.bit[1]=0) ? bfp_ZERO : bfp_CONVERT_FROM_BFLOAT16(VSR[32×BX+B].word[j].hword[1])
      reset_flags()
      p1 ← bfp_MULTIPLY(src11, src21)
      v1 ← bfp_MULTIPLY_ADD(src12, src22, p1)
      r1 ← bfp_ROUND_TO_BFP32_SGNF1AND(v1)
      if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
      if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
      if vxisi_flag=1 then SetFX(FPSCR.VXISI)
      if xx_flag=1 then SetFX(FPSCR.XX)
      if "[pm]xvb16ger2" then do
        reset_flags()
        r2 ← bfp_ROUND_TO_BFP32_NO_TRAP(r1)
        ACC[AT][i].word[j] ← bfp32_CONVERT_FROM_BFP(r2)
        if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
        if vxisi_flag=1 then SetFX(FPSCR.VXISI)
        if ox_flag=1 then SetFX(FPSCR.OX)
        if ux_flag=1 then SetFX(FPSCR.UX)
        if xx_flag=1 then SetFX(FPSCR.XX)
        end
      else do
        acc ← bfp_CONVERT_FROM_BFP32(ACC[AT][i].word[j])
        reset_flags()
        if "[pm]xvb16ger2pp" then v ← bfp_ADD(r1, acc)
        if "[pm]xvb16ger2pn" then v ← bfp_ADD(r1, bfp_NEGATE(acc))
        if "[pm]xvb16ger2np" then v ← bfp_ADD(bfp_NEGATE(r1), acc)
        if "[pm]xvb16ger2nn" then v ← bfp_ADD(bfp_NEGATE(r1), bfp_NEGATE(acc))
        r2 ← bfp_ROUND_TO_BFP32_NO_TRAP(v)
        ACC[AT][i].word[j] ← bfp32_CONVERT_FROM_BFP(r2)
        if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
        if vxisi_flag=1 then SetFX(FPSCR.VXISI)
        if ox_flag=1 then SetFX(FPSCR.OX)
        if ux_flag=1 then SetFX(FPSCR.UX)
        end
      end
    end
  end
end

end
Let \( X_A \) be the value of \( 32 \times X_A + A \). If \( X_A \) is in the range \( 4 \times AT \) to \( 4 \times AT + 3 \), the instruction form is invalid.

Let \( X_B \) be the value of \( 32 \times X_B + B \). If \( X_B \) is in the range \( 4 \times AT \) to \( 4 \times AT + 3 \), the instruction form is invalid.

Let the contents of \( ACC[AT] \) be a \( 4 \times 4 \) matrix of single-precision floating-point values.

For \( xvbf16ger2 \), \( xvbf16ger2pp \), \( xvbf16ger2pn \), \( xvbf16ger2np \), or \( xvbf16ger2nn \), let \( PMSK = 0b11 \), \( XMSK = 0b1111 \), and \( YMSK = 0b1111 \).

For each integer value \( i \) from 0 to 3, and each integer value \( j \) from 0 to 3, do the following.

If bit \( i \) of \( XMSK \) is equal to 1 and bit \( j \) of \( YMSK \) is equal to 1, do the following.

If bit 0 of \( PMSK \) is equal to 1, let \( src10 \) be the bfloat16 floating-point value in halfword 0 of word element \( i \) of \( VSR[XA] \) and let \( src20 \) be the bfloat16 floating-point value in halfword 0 of word element \( j \) of \( VSR[XB] \). Otherwise, let \( src10 \) be the value 0.0 and let \( src20 \) be the value 0.0, causing the product of \( src10 \) and \( src20 \) to be 0.0.

If bit 1 of \( PMSK \) is equal to 1, let \( src11 \) be the bfloat16 floating-point value in halfword 1 of word element \( i \) of \( VSR[XA] \) and let \( src21 \) be the bfloat16 floating-point value in halfword 1 of word element \( j \) of \( VSR[XB] \). Otherwise, let \( src11 \) be the value 0.0 and let \( src21 \) be the value 0.0, causing the product of \( src11 \) and \( src21 \) to be 0.0.

Let \( prod \) be the product of \( src10 \) and \( src20 \), having infinite precision and unbounded exponent range.

Let \( psum \) be the sum of the product, \( src11 \) multiplied by \( src21 \), and \( prod \), having infinite precision and unbounded exponent range.

Let \( r1 \) be the value \( psum \) with its significand rounded to 24-bit precision using the rounding mode specified by \( RN \), but retaining unbounded exponent range (i.e., cannot overflow or underflow).

For \( [pm]xvbf16ger2 \), do the following.

Let \( r2 \) be the value \( r1 \) rounded to 24-bit significand precision and 8-bit exponent range (i.e., single-precision) using the rounding mode specified by \( RN \).

\( r2 \) is placed into word element \( j \) of \( ACC[AT][i] \) in single-precision floating-point format.

For \( [pm]xvbf16ger2pp \), do the following.

Let \( r2 \) be the sum of \( r1 \) added to the single-precision floating-point value in word element \( j \) of \( ACC[AT][i] \), having infinite precision and unbounded exponent range.

Let \( r2 \) be the value \( v2 \) rounded to 24-bit significand precision and 8-bit exponent range (i.e., single-precision) using the rounding mode specified by \( RN \).

\( r2 \) is placed into word element \( j \) of \( ACC[AT][i] \) in single-precision floating-point format.

For \( [pm]xvbf16ger2pn \), do the following.

Let \( r2 \) be the sum of the negation of \( r2 \) added to the negation of the single-precision floating-point value in word element \( j \) of \( ACC[AT][i] \), having infinite precision and unbounded exponent range.

Let \( r2 \) be the value \( v2 \) rounded to 24-bit significand precision and 8-bit exponent range (i.e., single-precision) using the rounding mode specified by \( RN \).

\( r2 \) is placed into word element \( j \) of \( ACC[AT][i] \) in single-precision floating-point format.

For \( [pm]xvbf16ger2np \), do the following.

Let \( v2 \) be the sum of the negation of \( r2 \) added to the single-precision floating-point value in word element \( j \) of \( ACC[AT][i] \), having infinite precision and unbounded exponent range.

Let \( r3 \) be the value \( v3 \) rounded to 24-bit significand precision and 8-bit exponent range (i.e., single-precision) using the rounding mode specified by \( RN \).

\( r2 \) is placed into word element \( j \) of \( ACC[AT][i] \) in single-precision floating-point format.
For \([pm]\text{xvbf16ger2nn}\), do the following.

Let \(v_2\) be the sum of the negation of \(r_2\) added to the negation of the single-precision floating-point value in word element \(j\) of \(\text{ACC}[\text{AT}][i]\), having infinite precision and unbounded exponent range.

Let \(r_2\) be the value \(v_3\) rounded to 24-bit significand precision and 8-bit exponent range (i.e., single-precision) using the rounding mode specified by \(\text{RN}\).

\(r_2\) is placed into word element \(j\) of \(\text{ACC}[\text{AT}][i]\) in single-precision floating-point format.

Otherwise, the contents of word element \(j\) of \(\text{ACC}[\text{AT}][i]\) are set to \(0x0000_0000\).

Unlike other VSX Vector Floating-Point instructions, \(\text{ACC}[\text{AT}]\) is always updated by the execution of the instruction, even when a trap-enabled exception occurs. For every rounding operation that is performed as part of the execution of this instruction, if an exception occurs as the result of that particular rounding operation, the trap-disabled exception result is returned, even if that exception type is trap-enabled. Exception detection is based on the trap-disabled definition. Exception status is accumulated and the appropriate exception status bits in the FPSCR are updated at the completion of execution of the instruction. Otherwise, behavior is the same as any vector floating-point instruction that can cause an exception. Taking a Program interrupt on a trap-enabled exception when interrupts are enabled by MSR.FE0 and MSR.FE1 is still supported, albeit with the \(\text{ACC}[\text{AT}]\) updated based on a trap-disabled result.

**Special Registers Altered:**

\(FX\) \(VXSNAN\) \(VXIMZ\) \(VXISI\) \(OX\) \(UX\) \(XX\)

**Register Operand Data Layout for \([pm]\text{xvbf16ger2}\)[pp|pn|np|nn]**

<table>
<thead>
<tr>
<th>VSR[XA]</th>
<th>X[0][0]</th>
<th>X[0][1]</th>
<th>X[1][0]</th>
<th>X[1][1]</th>
<th>X[2][0]</th>
<th>X[2][1]</th>
<th>X[3][0]</th>
<th>X[3][1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC[AT][0]</td>
<td>T[0][0]</td>
<td>T[0][1]</td>
<td>T[0][2]</td>
<td>T[0][3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 1 63 24 86 48 09 61 1 2 1 2 7
Let $X$ be the $4\times 2$ matrix of bfloat16 floating-point values contained in $VSR[XA]$ in row-major format. Let $Y$ be the $4\times 2$ matrix of bfloat16 floating-point values contained in $VSR[XB]$ in row-major format. Let $ACC[AT]$ be the accumulator containing a $4\times 4$ matrix of single-precision floating-point values.

\[ \text{pmxvbf16ger2} \text{ performs the following form of accumulation of two outer products (rank 2 update).} \]

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 3:
\quad \text{ACC}[AT][i][j] = \text{fmadds}(X[i][1], Y[j][1], \text{fmulsx}(X[i][0], Y[j][0]))
\]

where \text{fmulsx}() is equivalent to a \text{fmuls} instruction that rounds the significand of its result to single-precision but retains an exponent having unbounded range.

\[ \text{pmxvbf16ger2pp} \text{ performs the following form of accumulation of two outer products (rank 2 update).} \]

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 3:
\quad \text{ACC}[AT][i][j] = \text{fadds}(\text{fmadds}(X[i][1], Y[j][1], \text{fmulsx}(X[i][0], Y[j][0])), \text{ACC}[AT][i][j])
\]

\[ \text{pmxvbf16ger2pn} \text{ performs the following form of accumulation of two outer products (rank 2 update).} \]

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 3:
\quad \text{ACC}[AT][i][j] = \text{fsubs}(\text{fmadds}(X[i][1], Y[j][1], \text{fmulsx}(X[i][0], Y[j][0])), \text{ACC}[AT][i][j])
\]

\[ \text{pmxvbf16ger2np} \text{ performs the following form of accumulation of two outer products (rank 2 update).} \]

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 3:
\quad \text{ACC}[AT][i][j] = \text{fadds}(\text{fnmadds}(X[i][1], Y[j][1], \text{fmulsx}(X[i][0], Y[j][0])), \text{ACC}[AT][i][j])
\]

\[ \text{pmxvbf16ger2nn} \text{ performs the following form of accumulation of two outer products (rank 2 update).} \]

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 3:
\quad \text{ACC}[AT][i][j] = \text{fsubs}(\text{fnmadds}(X[i][1], Y[j][1], \text{fmulsx}(X[i][0], Y[j][0])), \text{ACC}[AT][i][j])
\]
VSX Vector Compare Equal To Double-Precision XX3-form

\[
xvcmpeqdp \ XT,XA,XB \quad (Rc=0) \\
xvcmpeqdp. \ XT,XA,XB \quad (Rc=1)
\]

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>Rc</th>
<th>99</th>
<th>VSX Unavailable</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();

ex_flag ← 0b0
all_false ← 0b1
all_true ← 0b1

for i = 0 to 1
    reset_xflags();
    src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
    src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
    vxsnan_flag ← IsSNaN(src1) | IsSNaN(src2)
    if src1 = src2 then do
        vresult.dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
        all_false ← 0b0
    end
    else do
        vresult.dword[i] ← 0x0000_0000_0000_0000
        all_true ← 0b0
    end

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)

if ex_flag=0 then VSR[32×TX+T] ← vresult
if Rc=1 then do
    if vex_flag=0 then
        CR[6] ← all_true || 0b0 || all_false || 0b0
    else
        CR[6] ← 0bUUUUU
end

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is compared to src2.

The contents of doubleword element i of VSR[XT] are set to all 1s if src1 is equal to src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 is set to indicate all vector elements compared true.
- Bit 1 is set to 0.
- Bit 2 is set to indicate all vector elements compared false.
- Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR field 6 are undefined if Rc is equal to 1.

Special Registers Altered

CR field 6 ........................... (if Rc=1)
FX VXSNAN

VSR Data Layout for xvcmpeqdp[]

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>
VSX Vector Compare Equal To
Single-Precision XX3-form

\[ \text{xvcmpeqsp} \quad \text{XT, XA, XB} \quad \text{(Rc=0)} \]
\[ \text{xvcmpeqsp} \quad \text{XT, XA, XB} \quad \text{(Rc=1)} \]

Let XT be the value \( 32 \times TX + T \).
Let XA be the value \( 32 \times AX + A \).
Let XB be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 3, do the following.
Let src1 be the single-precision floating-point operand in word element \( i \) of VSR[XA].
Let src2 be the single-precision floating-point operand in word element \( i \) of VSR[XB].
src1 is compared to src2.
The contents of word element \( i \) of VSR[XT] are set to all 1s if src1 is equal to src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.
Two zero inputs of same or different signs return true for that element.
Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.
– Bit 0 is set to indicate all vector elements compared true.
– Bit 1 is set to 0.
– Bit 2 is set to indicate all vector elements compared false.
– Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT], and the contents of CR field 6 are undefined if Rc is equal to 1.

Special Registers Altered
CR field 6 ......................... (if Rc=1)
FX VXSNAN

\[ \begin{array}{cccccccc}
\end{array} \]

VSR Data Layout for xvcmpeqsp[]

|------|-----------------|-----------------|-----------------|-----------------|
VSX Vector Compare Greater Than or Equal To Double-Precision XX3-form

xvcmpgedp XT,XA,XB (Rc=0)
xvcmpgedp XT,XA XB (Rc=1)

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>Rc</th>
<th>115</th>
<th>116</th>
<th>117</th>
<th>118</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>60</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>22</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0
all_false ← 0b1
all_true ← 0b1

for i = 0 to 1 do
  reset_xflags()
  src1 ← bfp_CONVERT_FROM_BFP64(VSR[XA].dword[i])
  src2 ← bfp_CONVERT_FROM_BFP64(VSR[XB].dword[i])
  if src1.class.SNaN | src2.class.SNaN then
    vxsnan_flag ← 0b1
    if FPSCR.VE=0 then vxvc_flag ← 0b1
  end
  else
    vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)
  end
  if src1 >= src2 then
    vresult.dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
    all_false ← 0b0
  else
    vresult.dword[i] ← 0x0000_0000_0000_0000
    all_true ← 0b0
  end
  if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
  if vxvc_flag=1 then SetFX(FPSCR.VXVC)
  ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag) | (FPSCR.VE & vxvc_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
if Rc=1 then do
  if vex_flag=0 then
    CR.field[6] ← all_true || 0b0 || all_false || 0b0
  else
    CR.field[6] ← 0bUUUU
end

Let XT be the value 32×TX + T.
Let XA be the value 32×XA + A.
Let XB be the value 32×XB + B.

For each integer value i from 0 to 1, do the following.
Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is compared to src2.

The contents of doubleword element i of VSR[XT] are set to all 1s if src1 is greater than or equal to the double-precision floating-point operand in doubleword element i of src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.
− Bit 0 is set to indicate all vector elements compared true.
− Bit 1 is set to 0.
− Bit 2 is set to indicate all vector elements compared false.
− Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR field 6 are undefined if Rc is equal to 1.

Special Registers Altered
CR field 6 ........................................ (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgedp[]

Let XT be the value 32×TX + T.
Let XA be the value 32×XA + A.
Let XB be the value 32×XB + B.

For each integer value i from 0 to 1, do the following.
Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is compared to src2.

The contents of doubleword element i of VSR[XT] are set to all 1s if src1 is greater than or equal to the double-precision floating-point operand in doubleword element i of src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.
− Bit 0 is set to indicate all vector elements compared true.
− Bit 1 is set to 0.
− Bit 2 is set to indicate all vector elements compared false.
− Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR field 6 are undefined if Rc is equal to 1.

Special Registers Altered
CR field 6 ........................................ (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgedp[]

src1 VSR[XA].dword[0] VSR[XA].dword[1]
src2 VSR[XB].dword[0] VSR[XB].dword[1]
tgt VSR[XT].dword[0] VSR[XT].dword[1]
**VSX Vector Compare Greater Than or Equal To Single-Precision XX3-form**

\[
\text{xvcmpgesp}: \quad XT,XA,XB \quad (Rc=0) \\
\text{xvcmpgesp}: \quad XT,XA,XB \quad (Rc=1)
\]

Let \( XT \) be the value \( 32 \times TX + T \).  
Let \( XA \) be the value \( 32 \times AX + A \).  
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 3, do the following.

Let \( src1 \) be the single-precision floating-point operand in word element \( i \) of \( VSR[XA] \).

Let \( src2 \) be the single-precision floating-point operand in word element \( i \) of \( VSR[XB] \).

\( src1 \) is compared to \( src2 \).

The contents of word element \( i \) of \( VSR[XT] \) are set to all 1s if \( src1 \) is greater than or equal to \( src2 \), and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If \( Rc=1 \), CR Field 6 is set as follows.

- Bit 0 is set to indicate all vector elements compared true.
- Bit 1 is set to 0.
- Bit 2 is set to indicate all vector elements compared false.
- Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( VSR[XT] \) and the contents of CR field 6 are undefined if \( Rc \) is equal to 1.

**Special Registers Altered**

| CR Field 6 | FX, VXSNAN, VXVC |

**VSR Data Layout for xvcmpgesp[]**

|------|----------------|----------------|----------------|----------------|
VSX Vector Compare Greater Than Double-Precision XX3-form

\[
xvcmpgtdp \quad XT,XA,XB \quad (Rc=0)
\]
\[
xvcmpgtdp. \quad XT,XA,XB \quad (Rc=1)
\]

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is compared to src2.

The contents of doubleword element i of VSR[XT] are set to all 1s if src1 is greater than src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return false for that element.

If Rc = 1, CR Field 6 is set as follows.
- Bit 0 is set to indicate all vector elements compared true.
- Bit 1 is set to 0.
- Bit 2 is set to indicate all vector elements compared false.
- Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR field 6 are undefined if Rc is equal to 1.

Special Registers Altered
CR field 6 . . . . . . . . . . . . . . . . . . . . . . . . (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgtdp[.]
VSX Vector Compare Greater Than Single-Precision XX3-form

```plaintext
xvcmpgtsp XT,XA,XB (Rc=0)
xvcmpgtsp XT,XA,XB (Rc=1)
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.
Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
Let src2 be the single-precision floating-point operand in word element i of VSR[XB].
src1 is compared to src2.
The contents of word element i of VSR[XT] are set to all 1s if src1 is greater than src2, and is set to all 0s otherwise.
A NaN input causes the comparison to return false for that element.
Two zero inputs of same or different signs return false for that element.

If Rc=1, CR Field 6 is set as follows.
- Bit 0 is set to indicate all vector elements compared true.
- Bit 1 is set to 0.
- Bit 2 is set to indicate all vector elements compared false.
- Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR field 6 are undefined if Rc is equal to 1.

Special Registers Altered
CR field 6
FX VXSNAN VXVC

VSR Data Layout for xvcmpgtsp[]
```plaintext
|-------|----------------|----------------|----------------|----------------|
```

VSX Vector Copy Sign Double-Precision

**XX3-form**

```plaintext
VSX Vector Copy Sign Double-Precision

xvcpsgndp  XT,XA,XB
```

**VSR Data Layout for xvcpsgnlp**

- **src1**: VSR[XA].dword[0]
- **src2**: VSR[XB].dword[0]
- **tgt**: VSR[XT].dword[0]

**Special Registers Altered**

None

**Extended Mnemonics**

- **Extended mnemonic**: vvcpsgnlp
  - **Equivalent to**: vvcpsgndp

**VSR Data Layout for xvcpsgnsp**


**Special Registers Altered**

None

**Extended Mnemonics**

- **Extended mnemonic**: vvcpsgnsn
  - **Equivalent to**: vvcpsgndp
VSX Vector Convert bfloat16 to Single-Precision format XX2-form

\( \text{xvcvbf16sp XT XB} \)

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>16</th>
<th>B</th>
<th>475</th>
<th>B[TX]</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

If MSR.VSX=0 then VSX_Unavailable().

reset_flags() 

do i = 0 to 3

VSR[32\times TX + T].word[i].hw[0] \leftarrow VSR[32\times BX + B].word[i].hw[1]
VSR[32\times TX + T].word[i].hw[1] \leftarrow 0x0000
end

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 3, do the following.

The contents of the rightmost halfword of word element \( i \) of VSR[\( XB \)] are placed into the leftmost halfword of word element \( i \) of VSR[\( XT \)].

The contents of the rightmost halfword of word element \( i \) of VSR[\( XT \)] are set to 0.

Special Registers Altered:
None

Register Operand Data Layout for xvcvhpsp

<table>
<thead>
<tr>
<th>src2</th>
<th>tgt</th>
<th>unused</th>
<th>unused</th>
<th>unused</th>
<th>unused</th>
<th>unused</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
<td>64</td>
<td>80</td>
<td>90</td>
<td>112</td>
</tr>
</tbody>
</table>
Let XT be the value $32 \times TX + T$.
Let XB be the value $32 \times BX + B$.

For each integer value $i$ from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element $i$ of VSR[XB].

src is rounded to single-precision using the rounding mode specified by RN.

The result is placed into bits 0:31 and bits 32:63 of doubleword element $i$ of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

### Special Registers Altered

FX OX UX XX VXSNAN

### Programming Note

Previous versions of the architecture allowed the contents of bits 32:63 of each doubleword in the result register to be undefined, however, all processors that support this instruction write the result into bits 32:63 of each doubleword in the result register as well as into bits 0:31, as is required by this version of the architecture.

### Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.
VSX Vector Convert with round to zero
Double-Precision to Signed Doubleword
format XX2-form

\[
\text{vxcvdpsxds} \quad \text{XT,XB}
\]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>H</th>
<th>B</th>
<th>472</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- if MSR.VSX=0 then VSX_Unavailable()
- \(ex\_flag \leftarrow 0b0\)
- do \(i = 0\) to 1
  - reset_xflags()
  - \(src \leftarrow \text{bfp}_6 \text{CONVERT FROM BFP64(} \text{VSR[32×BX+B]}.\text{dword}[i]\text{)}\)
  - \(\text{rnd} \leftarrow \text{bfp}_6 \text{ROUND TO_INTEGER_TRUNC}(src)\)
  - \(\text{vresult.dword}[i] \leftarrow \text{s64}_6 \text{CONVERT FROM BFP}$(\text{rnd})$
  - if \(\text{vsnan_flag}=1\) then SetFX(FPSCR.VXSNAN)
  - if \(\text{vxcvi_flag}=1\) then SetFX(FPSCR.VXCVI)
  - if \(\text{xx_flag}=1\) then SetFX(FPSCR.XX)
  - \(ex\_flag \leftarrow ex\_flag | (\text{FPSCR.VE & vsnan_flag})\)
  - \( | (\text{FPSCR.VE & vxcvi_flag})\)
  - \( | (\text{FPSCR.XE & xx_flag})\)
- end
- if \(ex\_flag=0\) then \(\text{VSR[XT]} \leftarrow \text{result}\)

Let \(\text{XT}\) be the value \(32×\text{TX} + T\).
Let \(\text{XB}\) be the value \(32×\text{BX} + B\).

For each integer value \(i\) from 0 to 1, do the following.
Let \(src\) be the double-precision floating-point operand in doubleword element \(i\) of \(\text{VSR[XB]}\).

- If \(src\) is a NaN, the result is the value \(0x8000_0000_0000_0000\) and \(\text{VXCVI}\) is set to 1.
- If \(src\) is an SNaN, \(\text{VXSNAN}\) is also set to 1.
- Otherwise, \(src\) is rounded to a floating-point integer using the rounding mode \text{Round Toward Zero}.

If the rounded value is greater than \(2^{63}-1\), the result is \(0x7FFF_FFFF_FFFF_FFFF\) and \(\text{VXCVI}\) is set to 1.

Otherwise, if the rounded value is less than \(-2^{63}\), the result is \(0x8000_0000_0000_0000\) and \(\text{VXCVI}\) is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and if the result is inexact (i.e., not equal to \(src\)), \(XX\) is set to 1.

The result is placed into doubleword element \(i\) of \(\text{VSR[XT]}\).

See Table 116.

If a trap-enabled exception occurs in any element of the vector, no results are written to \(\text{VSR[XT]}\).

Special Registers Altered
\(\text{FX XX VXSNAN VXCVI}\)

Programming Note
\(\text{vxcvdpsxds}\) rounds using \text{Round towards Zero} rounding mode. For other rounding modes, software must use a \text{Round to Double-Precision Integer} instruction that corresponds to the desired rounding mode, including \(\text{xvrdpic}\) which uses the rounding mode specified by the RN.
FPSCR.CE | FPSCR.XE | Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>FPSCR.CE</th>
<th>FPSCR.XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin−1</td>
<td>0</td>
<td>–</td>
<td>T(Nmin, f[x(VXCVI)]</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>f[x(VXCVI)], error()</td>
</tr>
<tr>
<td>Nmin−1 &lt; src &lt; Nmin</td>
<td>0</td>
<td>yes</td>
<td>T(Nmin, f[x(XR)]</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>f[x(XR)], error()</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>–</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>–</td>
<td>no</td>
<td>T(2i(trunc(src)))</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>yes</td>
<td>T(2i(trunc(src))), f[x[XR]]</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>f[x(XR)], error()</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>–</td>
<td>–</td>
<td>no</td>
</tr>
</tbody>
</table>

Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.

| src | Nmax+1 | – | T(Nmax), f[x(VXCVI)] |
|     | 1 | – | f[x(VXCVI)], error() |
| src is a QNaN | – | – | T(Nmin, f[x(VXCVI)] |
|     | 1 | – | f[x(VXCVI), fx(VXSNAN)] |
| src is a SNaN | – | – | T(Nmin, f[x(VXCVI)], f[x(VXSNAN)], error()) |

Explanation:

- **error()** The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode.
- **Update of VSR[XT]** is suppressed.
- **f2i(x)** The double-precision floating-point integer value x is converted to 64-bit signed integer format.
- **fx(x)** FPSCR.FX is set to 1 if FPSCR.x=0.
- **Nmin** The smallest signed integer doubleword value, -263 (0x8000_0000_0000_0000).
- **Nmax** The largest signed integer doubleword value, 263−1 (0x7FFF_FFFF_FFFF_FFFF).
- **src** The double-precision floating-point value in doubleword element i of VSR[XB] (where i={0,1}).
- **T(x)** The signed integer doubleword value i is placed in doubleword element i of VSR[XT] (where i={0,1}).
- **trunc(x)** The double-precision floating-point value x is truncated to a floating-point integer.

Table 116. Actions for xcvdpsxd
VSX Vector Convert with round to zero
Double-Precision to Signed Word format
XX2-form

\texttt{xvcvdpsxws XT,XB}

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>11</th>
<th>B</th>
<th>216</th>
<th>B[T]</th>
</tr>
</thead>
</table>

If the rounded value is less than \(2^{31}\), the result is \(0x0000_0000\) and VXCVI is set to 1.

Otherwise, if the rounded value is greater than \(2^{31}\cdot1\), the result is \(0x8000_0000\) and VXSNAN is set to 1.

otherwise, the result is the rounded value converted to 32-bit signed-integer format, and if the result is exact (i.e., not equal to \(\text{src}\)), XX is set to 1.

The result is placed into bits 0:31 of doubleword element \(i\) of \(\text{VSR}[XT]\).

The result is also placed into bits 32:63 of doubleword element \(i\) of \(\text{VSR}[XT]\).

See Table 117.

If a trap-enabled exception occurs in any element of the vector, no results are written to \(\text{VSR}[XT]\).

**Special Registers Altered**

\(FX \ XX \ VXSNAN \ VXCVI\)

**Programming Note**

\texttt{xvcvdpsxws} rounds using Round towards Zero rounding mode. For other rounding modes, software must use a \texttt{Round to Double-Precision Integer} instruction that corresponds to the desired rounding mode, including \texttt{xvrdpic} which uses the rounding mode specified by RN.

**Programming Note**

Previous versions of the architecture allowed the contents of words 1 and 3 of the result register to be undefined. However, all processors that support this instruction write the result into words 0 and 1 and words 2 and 3 of the result register, as is required by this version of the architecture.
<table>
<thead>
<tr>
<th>Returned Results and Status Setting</th>
<th>FPSCR[VE]</th>
<th>FPSCR[XE]</th>
<th>lineart? (funcmax = src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>src [ Nmin -1] 0 - -</td>
<td>T(Nmin), fx(VXCVI)</td>
<td>1 - -</td>
<td>T(Nmin), fx(VXCVI), error()</td>
</tr>
<tr>
<td>1 - -</td>
<td>T(Nmin), fx(VXCVI), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmin -1 &lt; src &lt; Nmin</td>
<td>- 0 yes</td>
<td>T(Nmin), fx(XX)</td>
<td>1 yes</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>- - no</td>
<td>T(Nmin)</td>
<td></td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>- 0 yes</td>
<td>T(2i(trunc src)), fx(XX)</td>
<td>1 yes</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>- - no</td>
<td>T(Nmax)</td>
<td></td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>- 0 yes</td>
<td>T(Nmax), fx(XX)</td>
<td>1 yes</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>- - no</td>
<td>T(Nmax)</td>
<td></td>
</tr>
<tr>
<td>Nmax+1 &lt; src &lt; Nmax+1</td>
<td>- 0 yes</td>
<td>T(Nmax), fx(VXCVI)</td>
<td>1 - -</td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0 - -</td>
<td>T(Nmin), fx(VXCVI)</td>
<td></td>
</tr>
<tr>
<td>1 - -</td>
<td>T(Nmin), fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**
- **error()** The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode.
- **(2i|x)** The double-precision floating-point integer value x is converted to 32-bit signed integer format.
- **fx(x)** FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
- **Nmin** The smallest signed integer word value, \(-2^{31}(0x8000_0000)\).
- **Nmax** The largest signed integer word value, \(2^{31}(0xFFFF_FFFF)\).
- **src** The double-precision floating-point value in doubleword element i of VSR[XB] (where i=0,1).
- **T(x)** The signed integer word value x is placed in word elements \(2xi\) and \(2xi+1\) of VSR[XT] (where i=0,1).
- **trunc(x)** The double-precision floating-point value x is truncated to a floating-point integer.

**Table 117. Actions for xvcvdpsxws**
**VSX Vector Convert with round to zero**

**Double-Precision to Unsigned Doubleword format XX2-form**

```
xvcvdpuxds XT,XB
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>T</th>
<th></th>
<th></th>
<th>B</th>
<th>456</th>
<th>B1</th>
<th>B0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>30</td>
</tr>
</tbody>
</table>

- If MSR.VSX=0 then VSX_Unavailable()
- \text{ex} \_\text{flag} \leftarrow 0b0
- do \text{i} = 0 to 1
  - \text{reset}_\text{flags}()
  - \text{src} \leftarrow \text{bfp}_\text{CONVERT}_\text{FROM}_\text{BFP64} (\text{VSR}\[32\times\text{BX}+\text{B}.\text{dword}[\text{i}])
  - \text{rnd} \leftarrow \text{bfp}_\text{ROUND}_\text{TO}_\text{INTEGER}_\text{TRUNC} (\text{src})
  - \text{vresult}.\text{dword}[\text{i}] \leftarrow \text{ui64}_\text{CONVERT}_\text{FROM}_\text{BFP} (\text{rnd})
- if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
- if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
- if xx_flag=1 then SetFX(FPSCR.XX)
- \text{ex} \_\text{flag} \leftarrow \text{ex} \_\text{flag} | (FPSCR.VE & vxsnan_flag)
- \text{lag} \leftarrow \text{ex} \_\text{flag} | (FPSCR.VE & vxcvi_flag)
- end
- if \text{ex} \_\text{flag}=0 then VSR[32\times\text{TX}+\text{T}] \leftarrow \text{vresult}

Let \text{XT} be the value \(32\times\text{TX} + \text{T}\).
Let \text{XB} be the value \(32\times\text{BX} + \text{B}\).

For each integer value \text{i} from 0 to 1, do the following.
Let \text{src} be the double-precision floating-point operand in doubleword element \text{i} of VSR[\text{XB}].

- If \text{src} is a NaN, the result is the value \(0x0000_0000_0000_0000\) and VXCVI is set to 1. If \text{src} is an SNaN, VXSNAN is also set to 1.
- Otherwise, \text{src} is rounded to a floating-point integer using the rounding mode Round Toward Zero.
- If the rounded value is greater than \(2^{64} - 1\), the result is \(0xffffffff\) and VXCVI is set to 1.
- Otherwise, if the rounded value is less than 0, the result is \(0x0000_0000_0000_0000\) and VXCVI is set to 1.
- Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format, and if the result is inexact (i.e., not equal to \text{src}), XX is set to 1.

The result is placed into doubleword element \text{i} of VSR[\text{XT}].

See Table 118.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[\text{XT}].

**Special Registers Altered**

- FX
- XX
- VXSNAN
- VXCVI

```
Programming Note
```

\text{xvcvdpuxds} rounds using Round towards Zero rounding mode. For other rounding modes, software must use a Round to Double-Precision Integer instruction that corresponds to the desired rounding mode, including \text{xvrdpic} which uses the rounding mode specified by the RN.

```
VSR Data Layout for xvcvdpuxds
```

```
tgt
src
```

\text{src} \quad \text{VSR}[\text{XB}].\text{dword}[0] \quad \text{VSR}[\text{XB}].\text{dword}[1]
\text{tgt} \quad \text{VSR}[\text{XT}].\text{dword}[0] \quad \text{VSR}[\text{XT}].\text{dword}[1]
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>NE</th>
<th>NEq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin–1</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>fx(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin–1 &lt; src &lt; Nmin</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td>0</td>
<td>yes</td>
<td>T(ceil(trunc(src))), fx(XX)</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>T(Nmax), fx(XX), error()</td>
</tr>
<tr>
<td>src &gt; Nmax+1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

### Explanation:

- `error()` The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
- Update of VSR[XT] is suppressed.
- `f2i(x)` The double-precision floating-point integer value `x` is converted to 64-bit unsigned integer format.
- `fx(x)` FPSCR.FX is set to 1 if `FPSCR.x=0`. `FPSCR.x` is set to 1.
- `Nmin` The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
- `Nmax` The largest unsigned integer doubleword value, 2^64–1 (0xFFFF_FFFF_FFFF_FFFF).
- `src` The double-precision floating-point value in doubleword element `i` of VSR[XX] (where `i=0,1`).
- `T(x)` The unsigned integer doubleword value `x` is placed in doubleword element `i` of VSR[XT] (where `i=0,1`).
- `trunc(x)` The double-precision floating-point value `x` is truncated to a floating-point integer.

### Table 118. Actions for vxvcudpuxds
### VSX Vector Convert with round to zero

**Double-Precision to Unsigned Word format**

**XX2-form**

\[
\text{xvcvdpuxws} \quad \text{XT, XB}
\]

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>200</th>
<th>B (bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>III</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If MSR.VSX=0 then VSX_Unavailable()
- \( \text{ex\_flag} \leftarrow 0b0 \)
- \( \text{do } i = 0 \text{ to } 1 \)
- \( \text{reset\_xflags}() \)
- \( \text{src} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP64(VSR[32\times BX+B].dword[i])} \)
- \( \text{rnd} \leftarrow \text{bfp\_ROUND\_TO\_INTEGER\_TRUNC(src)} \)
- \( \text{vresult.dword[i].word[0]} \leftarrow \text{ui32\_CONVERT\_FROM\_BFP(rnd)} \)
- \( \text{vresult.dword[i].word[1]} \leftarrow \text{ui32\_CONVERT\_FROM\_BFP(rnd)} \)
- \( \text{if vxsnan\_flag=1 then SetFX(FPSCR.VXSNAN)} \)
- \( \text{if vxcvi\_flag=1 then SetFX(FPSCR.VXCVI)} \)
- \( \text{if xx\_flag=1 then SetFX(FPSCR.XX)} \)
- \( \text{ex\_flag} \leftarrow \text{ex\_flag} | (FPSCR.VE \& vxsnan\_flag) \)
- \( \quad | (FPSCR.VE \& vxcvi\_flag) \)
- \( \quad | (FPSCR.XE \& xx\_flag) \)
- \( \text{end} \)
- \( \text{if ex\_flag=0 then VSR[32\times TX+T] \leftarrow \text{vresult}} \)

Let \( \text{XT} \) be the value \( 32\times TX + T \).

Let \( \text{XB} \) be the value \( 32\times BX + B \).

For each integer value \( i \) from 0 to 1, do the following.

Let \( \text{src} \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[XB].

- If \( \text{src} \) is a NaN, the result is the value \( 0x8000_0000 \) and VXCVI is set to 1. If \( \text{src} \) is an SNaN, VXSNAN is also set to 1.

Otherwise, \( \text{src} \) is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than \( 2^{32}-1 \), the result is \( 0xFFFF_FFFF \) and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is \( 0x0000_0000 \) and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and if the result is inexact (i.e., not equal to \( \text{src} \)), XX is set to 1.

The result is placed into bits 0:31 of doubleword element \( i \) of VSR[XT].

The result is also placed into bits 32:63 of doubleword element \( i \) of VSR[XT].

See Table 119.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

#### Special Registers Altered

FX XX VXSNAN VXCVI

---

**Programming Note**

\( \text{xvcvdpuxws} \) rounds using Round towards Zero rounding mode. For other rounding modes, software must use a Round to Double-Precision Integer instruction that corresponds to the desired rounding mode, including \( \text{xvrdpic} \) which uses the rounding mode specified by RN.

---

**Programming Note**

Previous versions of the architecture allowed the contents of words 1 and 3 of the result register to be undefined. However, all processors that support this instruction write the result into words 0 and 1 and words 2 and 3 of the result register, as is required by this version of the architecture.

---

### VSR Data Layout for xvcvdpuxws

<table>
<thead>
<tr>
<th></th>
<th>VSR[XB].dword[0]</th>
<th>VSR[XB].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 119. Actions for xvcvdpxwxs

<table>
<thead>
<tr>
<th>src</th>
<th>NE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin-1</td>
<td>0</td>
<td>T(Nmin), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>fx(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>0</td>
<td>yes T(Nmin), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>-</td>
<td>no T(Nmin)</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>-</td>
<td>no T(N2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>yes T(N2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>-</td>
<td>no T(Nmax)</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>-</td>
<td>no T(Nmax)</td>
</tr>
<tr>
<td>src m Nmax+1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no T(VXCVI), error()</td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no T(VXCVI), fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

Explanation:

- error() The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
- Update of VSR[XT] is suppressed.
- \( f2i(x) \) The double-precision floating-point integer value \( x \) is converted to 32-bit unsigned integer format.
- \( fx(x) \) FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
- \( N_{\text{min}} \) The smallest unsigned integer word value, \( 0 (0x0000_0000) \).
- \( N_{\text{max}} \) The largest unsigned integer word value, \( 2^{32} - 1 (0xFFFF_FFFF) \).
- \( src \) The double-precision floating-point value \( x \) is placed in word elements \( 2^i \) and \( 2^i+1 \) of VSR[XT] (where \( i = 0, 1 \)).
- \( T(x) \) The double-precision floating-point value \( x \) is truncated to a floating-point integer.
VSX Vector Convert bfloat16 to Single-Precision format XX2-form

Let \(\text{XT}\) be the value \(32 \times \text{T} + T\).
Let \(\text{XB}\) be the value \(32 \times \text{BX} + B\).

For each integer value \(i\) from 0 to 3, do the following.
Let \(\text{src}\) be the half-precision floating-point value in the rightmost halfword of word element \(i\) of \(\text{VSR[XB]}\).

- If \(\text{src}\) is an SNaN, the result is the single-precision representation of that SNaN converted to a QNaN.
- Otherwise, if \(\text{src}\) is a QNaN, the result is the single-precision representation of that QNaN.
- Otherwise, if \(\text{src}\) is an Infinity, the result is the single-precision representation of Infinity with the same sign as \(\text{src}\).
- Otherwise, if \(\text{src}\) is a Zero, the result is the single-precision representation of Zero with the same sign as \(\text{src}\).
- Otherwise, if \(\text{src}\) is a denormal value, the result is the normalized single-precision representation of \(\text{src}\).
- Otherwise, the result is the single-precision representation of \(\text{src}\).

The result is placed into word element \(i\) of \(\text{VSR[XT]}\).

If a trap-enabled exception occurs, \(\text{VSR[XT]}\) is not modified.

**Special Registers Altered:**
\[FX \ VXS\text{SNAN}\]
VSX Vector Convert with round
Single-Precision to bfloat16 format XX2-form

**xvcvspbf16**  
**XT, XB**

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>17</th>
<th>B</th>
<th>475</th>
<th>BX</th>
<th>DX</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable||
do i = 0 to 3
  reset_flags()
  src ← bfp_CONVERT_FROM_BFP32(VSR[XB].word[i])
  rnd ← bfp_ROUND_TO_BFLOAT16_NO_TRAP(src)
  result.word[i].hword[0] ← 0x0000
  result.word[i].hword[1] ← bfloat16_CONVERT_FROM_BFP(rnd)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)
ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                 | (FPSCR.OE & ox_flag)
                 | (FPSCR.UE & ux_flag)
                 | (FPSCR.XE & xx_flag)
end
if ex_flag=0 then VSR[32×TX+T] ← result

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point value in word element i of VSR[XB].

If src is an SNaN, let result be the bfloat16 representation of that SNaN converted to a QNaN.

Otherwise, if src is a QNaN, let result be the bfloat16 representation of that QNaN.

Otherwise, if src is an Infinity, let result be the bfloat16 representation of Infinity with the same sign as src.

Otherwise, if src is a Zero, let result be the bfloat16 representation of Zero with the same sign as src.

Otherwise, let result be the bfloat16 representation of src rounded to bfloat16 precision using the rounding mode specified in RN.

result is placed into rightmost halfword of word element i of VSR[XT].

The leftmost halfword of word element i of VSR[XT] is set to 0x0000.

If a trap-enabled exception occurs, VSR[XT] is not modified.

**Special Registers Altered:**
FX VXSNAN OX UX XX

---

Register Operand Data Layout for xvcvspbf16

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>80</td>
<td>90</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>127</td>
<td>127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

850  Power ISA™ I
**VSX Vector Convert Single-Precision to Double-Precision format XX2-form**

\[
\text{xvcvspdp XT,XB}
\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>B</th>
<th>457</th>
<th>B17</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

\[
ex\_flag \leftarrow 0b0
\]

\[
do \ i = 0 \ to \ 1
\]

\[
\text{reset_xflags()}
\]

\[
src \leftarrow \text{bfp\_CONV\_FROM\_BFP32(VSR[BX×32+B].dword[i].word[0])}
\]

\[
vresult.dword[i] \leftarrow \text{bfp64\_CONV\_FROM\_BFP(src)}
\]

\[
\text{if vsnann_flag=1 then SetFX(FPSCR.VXSNAN)}
\]

\[
ex\_flag \leftarrow ex\_flag | (FPSCR.VE \& vsnann_flag)
\]

\[
\text{end}
\]

if ex_flag=0 then VSR[32×TX+T] \leftarrow vresult

Let XT be the value \(32\times TX + T\).

Let XB be the value \(32\times BX + B\).

For each integer value \(i\) from 0 to 1, do the following.

Let src be the single-precision floating-point operand in bits 0:31 of doubleword element \(i\) of VSR[XB].

src is placed into doubleword element \(i\) of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

FX VXSNAN

**VSR Data Layout for xvcvspdp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].word[0]</th>
<th>unused</th>
<th>VSR[XB].word[2]</th>
<th>unused</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tgt</th>
<th>VSR[XT].dword[0]</th>
<th>VSR[XT].dword[1]</th>
</tr>
</thead>
</table>

0 | 32 | 64 | 96 | 127 |
VSX Vector Convert with round
Single-Precision to bfloat16 format XX2-form

\[
xvcsvphp \quad XT, XB
\]

If \( src \) is an SNaN, the result is the half-precision representation of that SNaN converted to a QNaN.

Otherwise, if \( src \) is a QNaN, the result is the half-precision representation of that QNaN.

Otherwise, if \( src \) is an Infinity, the result is the half-precision representation of Infinity with the same sign as \( src \).

Otherwise, if \( src \) is a Zero, the result is the half-precision representation of Zero with the same sign as \( src \).

Otherwise, the result is the half-precision representation of \( src \) rounded to half-precision using the rounding mode specified by \( RN \).

The result is zero-extended and placed into word element \( i \) of \( VSR(NT) \).

If a trap-enabled exception occurs, \( VSR(NT) \) is not modified.

Special Registers Altered:
\( FX\ VXS\NaN\ OX\ UX\ XX\ )

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 3, do the following.
Let \( src \) be the single-precision floating-point value in word element \( i \) of \( VSR(XB) \).

VSR Data Layout for xvcsvphp

\[
\begin{array}{|c|c|c|c|c|}
\hline
\hline
0 & 16 & 32 & 48 & 64 & 80 & 96 & 112 & 127 \\
\end{array}
\]
VSX Vector Convert with round to zero
Single-Precision to Signed Doubleword
format \textit{XX2-form}

\texttt{xvcvspsxds XT,XB}

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>T</th>
<th>H</th>
<th>B</th>
<th>408</th>
<th>B(TX+T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\begin{verbatim}
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
  reset_xflags()
  src ← bfp_CONVERT_FROM_BFP32(VSR[XB].word[i].word[0])
  rnd ← bfp_ROUND_TO_INTEGER_TRUNC(src)
  vresult.dword[i] ← si64_CONVERT_FROM_BFP(rnd)
  if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
  if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
  if xx_flag=1 then SetFX(FPSCR.XX)
  ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
  lag ← ex_flag | (FPSCR.VE & vxcvi_flag)
  lag ← ex_flag | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
\end{verbatim}

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 1, do the following.

Let \( src \) be the single-precision floating-point operand \( i \times 2 \) of \( VSR[XB] \).

If \( src \) is a NaN, the result is the value \( 0x8000_0000_0000_0000 \) and \( VXCVI \) is set to 1. If \( src \) is an SNaN, \( VXSNAN \) is also set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than \( 2^{63} \cdot 1 \), the result is \( 0x7FFF_FFFF_FFFF_FFFF \) and \( VXCVI \) is set to 1.

Otherwise, if the rounded value is less than \( -2^{63} \), the result is \( 0x8000_0000_0000_0000 \) and \( VXCVI \) is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and if the result is inexact (i.e., not equal to \( src \)), \( XX \) is set to 1.

The result is placed into doubleword element \( i \) of \( VSR[XT] \).

See Table 119.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( VSR[XT] \).

\textbf{Special Registers Altered}

\begin{tabular}{l}
FX \quad XX \quad VXSNAN \quad VXCVI
\end{tabular}

\textbf{Programming Note}

\texttt{xvcvspsxds} rounds using Round towards Zero rounding mode. For other rounding modes, software must use a \textit{Round to Single-Precision Integer} instruction that corresponds to the desired rounding mode, including \texttt{xvrsptic} which uses the rounding mode specified by \( RN \).

\textbf{VSR Data Layout for xvcvspsxds}

\begin{tabular}{|c|c|c|c|c|}
\hline
\texttt{src} & \multicolumn{3}{c|}{\texttt{VSR[XB].word[0]}} & \texttt{VSR[XB].word[2]} & \texttt{unused} \\
\hline
\texttt{tgt} & \multicolumn{2}{c|}{\texttt{VSR[XT].dword[0]}} & \multicolumn{2}{c|}{\texttt{VSR[XT].dword[1]}} & \texttt{unused} \\
\hline
\end{tabular}
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>NE</th>
<th>InvExact?</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin-1</td>
<td>0</td>
<td>–</td>
<td>T(Nmin), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>T(Nmin), fx(VXCVI), error</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>0</td>
<td>yes</td>
<td>T(Nmin), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>T(XX), error</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>–</td>
<td>no</td>
<td>T(Nmin)</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>–</td>
<td>no</td>
<td>T(f2i(trunc(src))), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>T(XX), error</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>–</td>
<td>no</td>
<td>T(Nmax)</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>–</td>
<td>no</td>
<td>T(Nmax), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>T(XX), error</td>
</tr>
<tr>
<td>src = Nmax+1</td>
<td>0</td>
<td>–</td>
<td>T(Nmax), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>T(VXCVI), error</td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>–</td>
<td>T(Nmin), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>T(XX), error</td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0</td>
<td>–</td>
<td>T(Nmin), fx(VXCVI), fx(VXSnan)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>T(VXCVI), fx(VXSnan), error</td>
</tr>
</tbody>
</table>

**Explanation:**

- **error()** The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
- **f2i(x)** The single-precision floating-point integer value \( x \) is converted to 64-bit signed integer format.
- **fx(x)** FPSCR.x is set to 1 if FPSCR.x=0.
- **Nmin** The smallest signed integer doubleword value, \(-2^{63} \times 0x8000_0000_0000_0000\).
- **Nmax** The largest signed integer doubleword value, \(2^{63}-1 \times 0xFF_FFFF_FFFF_FFFF\).
- **src** The single-precision floating-point value in word element \( i \) of VSR[XB] (where \( i = (0, 2) \)).
- **T(x)** The signed integer doubleword value \( x \) is placed in doubleword element \( i \) of VSR[XT] (where \( i = (0, 1) \)).
- **trunc(x)** The single-precision floating-point value \( x \) is truncated to a floating-point integer.
VSX Vector Convert with round to zero
Single-Precision to Signed Word format
XX2-form

\textbf{xvcvspsxws XT,XB}

Let XT be the value $32 \times TX + T$. Let XB be the value $32 \times XB + B$.

For each integer value $i$ from 0 to 3, do the following.
Let src be the single-precision floating-point operand in word element $i$ of VSR[XB].

If src is a NaN, the result is the value $0x8000_0000$ and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Towards Zero.

If the rounded value is greater than $2^{31} - 1$, the result is $0x7FFF_FFFF$, and VXCVI is set to 1.

Otherwise, if the rounded value is less than $-2^{31}$, the result is $0x8000_0000$, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

The result is placed into word element $i$ of VSR[XT].

See Table 119.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

\textbf{Special Registers Altered}
FX XX VXSNAN VXCVI

\textbf{Programming Note}
\textit{xvcvspsxws} rounds using Round towards Zero rounding mode. For other rounding modes, software must use a \textit{Round to Single-Precision Integer} instruction that corresponds to the desired rounding mode, including \textit{xvrspsic} which uses the rounding mode specified by RN.

\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
\hline
\hline
\end{tabular}
Table 121. Actions for xvcvspsxws

<table>
<thead>
<tr>
<th>src</th>
<th>NE</th>
<th>NE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin-1</td>
<td>0</td>
<td>-</td>
<td>0 – T(Nmin), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1 – fx(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>-</td>
<td>yes</td>
<td>0 – T(Nmin), fx(XR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1 – fx(XR), error()</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>-</td>
<td>no</td>
<td>- T(Nmin)</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>-</td>
<td>no</td>
<td>- T(2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>-</td>
<td>no</td>
<td>- T(Nmax)</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>-</td>
<td>no</td>
<td>- T(Nmax), fx(XR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>src = Nmax+1</td>
<td>0</td>
<td>-</td>
<td>0 – T(Nmax), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1 – fx(VXCVI), error()</td>
</tr>
<tr>
<td>src = QNaN</td>
<td>0</td>
<td>-</td>
<td>0 – T(Nmin), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1 – fx(VXCVI), fx(VXSAN), error()</td>
</tr>
</tbody>
</table>

**Explanation:**

- **error()**: The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
- **Update of VSR[XT]** is suppressed.
- **fx(x)**: The single-precision floating-point integer value x is converted to 32-bit signed integer format.
- **f2i(x)**: If MSR.XF is set to 1 if MSKR.x=0, MSK.x is set to 1.
- **Nmin**: The smallest signed integer word value, $-2^{31}$ ($0x8000_0000$).
- **Nmax**: The largest signed integer word value, $2^{31}-1$ ($0x7FFF_FFFF$).
- **src**: The single-precision floating-point value in word element i of VSR[XB] (where i={0,1,2,3}).
- **T(x)**: The signed integer word value i is placed in word element i of VSR[XT] (where i={0,1,2,3}).
- **trunc(x)**: The single-precision floating-point value i is truncated to a floating-point integer.
VSX Vector Convert with round to zero
Single-Precision to Unsigned Doubleword format XX2-form

**xvcvspuxds XT, XB**

Let $XT$ be the value $32 \times TX + T$.
Let $XB$ be the value $32 \times BX + B$.

For each integer value $i$ from 0 to 1, do the following.
Let $src$ be the single-precision floating-point operand in word element $i \times 2$ of $VSR[XB]$.

If $src$ is a NaN, the result is the value $0x0000_0000_0000_0000$ and VXCVI is set to 1. If $src$ is an SNaN, VXSNAN is also set to 1.

Otherwise, $src$ is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{64}-1$, the result is $0xFFFFFFFF_FFFF_FFFF_FFFF$ and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is $0x0000_0000_0000_0000$ and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format, and if the result is inexact (i.e., not equal to $src$), XX is set to 1.

The result is placed into doubleword element $i$ of $VSR[XT]$.

See Table 119.

If a trap-enabled exception occurs in any element of the vector, no results are written to $VSR[XT]$.

**Special Registers Altered**

FX XX VXSNAN VXCVI

### Programming Note
**xvcvspuxds** rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Single-Precision Integer* instruction that corresponds to the desired rounding mode, including *xvrsptic* which uses the rounding mode specified by RN.

**VSR Data Layout for xvcvspuxds**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].word[0]</th>
<th>unused</th>
<th>VSR[XB].word[2]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>T</td>
<td>11</td>
<td>H</td>
<td>16</td>
</tr>
</tbody>
</table>
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>Nmin-1</th>
<th>0</th>
<th>-</th>
<th>-</th>
<th>T(Nmin), fx(VXCVI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>fx(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>0</td>
<td>yes</td>
<td>T(Nmin), fx(XR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fx(XR), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmin</td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>T(Nmin)</td>
<td></td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>-</td>
<td>no</td>
<td>T(f2i(trunc(src)))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>yes</td>
<td>T(f2i(trunc(src)), fx(XR))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fx(XR), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>-</td>
<td>0</td>
<td>yes</td>
<td>T(Nmax), fx(XR)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fx(XR), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmax</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmax), fx(VXCVI)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fx(VXCVI), fx(VXSNAN)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), fx(VXCVI), fx(VXSNAN)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

- **error()**
  - The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
  - Update of VSR[XT] is suppressed.
- **f2i(x)**
  - The single-precision floating-point integer value x is converted to 64-bit unsigned integer format.
- **fx(x)**
  - FPSCR.fx is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
- **Nmin**
  - The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
- **Nmax**
  - The largest unsigned integer doubleword value, 2^N-1 (0xFFFF_FFFF_FFFF_FFFF).
- **src**
  - The single-precision floating-point value in word element i of VSR[XB] (where i={0,2}).
- **T(x)**
  - The unsigned integer doubleword value x is placed in doubleword element i of VSR[XT] (where i={0,1}).
- **trunc(x)**
  - The single-precision floating-point value i is truncated to a floating-point integer.

### Table 122. Actions for xvcvspuxds
**VSX Vector Convert with round to zero**  
**Single-Precision to Unsigned Word format**  
**XX2-form**

Let $XT$ be the value $32 \times TX + T$.  
Let $XB$ be the value $32 \times BX + B$.

For each integer value $i$ from 0 to 3, do the following.  
Let $src$ be the single-precision floating-point operand in word element $i$ of $VSR[XB]$.

If $src$ is a NaN, the result is the value $0x0000_0000$ and $VXCVI$ is set to 1. If $src$ is an SNaN, $VXSNAN$ is also set to 1.

Otherwise, $src$ is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{32} - 1$, the result is $0xFFFF_{FFFF}$ and $VXCVI$ is set to 1.

Otherwise, if the rounded value is less than 0, the result is $0x0000_0000$ and $VXCVI$ is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integral format, and if the result is inexact (i.e., not equal to $src$), $XX$ is set to 1.

The result is placed into word element $i$ of $VSR[XT]$.

See Table 119.

If a trap-enabled exception occurs in any element of the vector, no results are written to $VSR[XT]$.

**Special Registers Altered**  
$FX \ XX \ VXSNAN \ VXCVI$

**Programming Note**  
`xvcvspuxws` rounds using Round towards Zero rounding mode. For other rounding modes, software must use a `Round to Single-Precision Integer` instruction that corresponds to the desired rounding mode, including `xvrspic` which uses the rounding mode specified by $RN$. 

---

<table>
<thead>
<tr>
<th>VSR Data Layout for xvcvspuxws</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>src</strong></td>
</tr>
<tr>
<td><strong>tgt</strong></td>
</tr>
</tbody>
</table>
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>N_min</th>
<th>N_max</th>
<th>Trunc(s)</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>N_min-1</td>
<td>1</td>
<td>-</td>
<td>T(N_min), F_x(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>N_min-1</td>
<td>1</td>
<td>-</td>
<td>f_x(VXCVI), error()</td>
</tr>
<tr>
<td>-</td>
<td>N_min-1</td>
<td>0</td>
<td>Trunc(s)</td>
<td>T(N_min), F_x(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>N_min-1</td>
<td>0</td>
<td>Trunc(s)</td>
<td>f_x(VXCVI), error()</td>
</tr>
<tr>
<td>-</td>
<td>N_min</td>
<td>-</td>
<td>-</td>
<td>T(N_min)</td>
</tr>
<tr>
<td>1</td>
<td>N_min</td>
<td>-</td>
<td>-</td>
<td>f_x(VXCVI), error()</td>
</tr>
<tr>
<td>-</td>
<td>N_min+1</td>
<td>-</td>
<td>-</td>
<td>T(N_max)</td>
</tr>
<tr>
<td>1</td>
<td>N_min+1</td>
<td>-</td>
<td>-</td>
<td>f_x(VXCVI), error()</td>
</tr>
<tr>
<td>0</td>
<td>N_min+1</td>
<td>0</td>
<td>Trunc(s)</td>
<td>T(N_max), F_x(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>N_min+1</td>
<td>0</td>
<td>Trunc(s)</td>
<td>f_x(VXCVI), error()</td>
</tr>
<tr>
<td>0</td>
<td>N_min+1</td>
<td>1</td>
<td>Trunc(s)</td>
<td>T(N_max), F_x(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>N_min+1</td>
<td>1</td>
<td>Trunc(s)</td>
<td>f_x(VXCVI), error()</td>
</tr>
</tbody>
</table>

Note: This case cannot occur as N_max is not representable in SP format but is included here for completeness.

### Explanation:

- **error()**: The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
- **Update of VSR[XT]** is suppressed.
- **f2i(x)**: The single-precision floating-point integer value x is converted to 32-bit unsigned integer format.
- **fx(x)**: FPSCH.FX is set to 1 if FPSCR.x=0. FPSCH.x is set to 1.
- **N_min**: The smallest unsigned integer word value, 0 (0x0000_0000).
- **N_max**: The largest unsigned integer word value, 2^53-1 (0xFFFF_FFFF).
- **src**: The single-precision floating-point value in word element i of VSR[XB] (where i={0,1,2,3}).
- **T(x)**: The unsigned integer word value x is placed in word element i of VSR[XT] (where i={0,1,2,3}).
- **trunc(x)**: The single-precision floating-point value x is truncated to a floating-point integer.

### Table 123: Actions for xvcvspuxws
**VSX Vector Convert with round Signed Doubleword to Double-Precision format XX2-form**

**xvcvsxddp**

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>504</th>
<th>D7-D0</th>
</tr>
</thead>
</table>

Let $XT$ be the value $32 \times TX + T$.
Let $XB$ be the value $32 \times BX + B$.

For each integer value $i$ from 0 to 1, do the following.
Let $src$ be the signed integer in doubleword element $i$ of $VSR[XB]$.

$s$ is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by $RN$.

The result is placed into doubleword element $i$ of $VSR[XT]$ in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to $VSR[XT]$.

**Special Registers Altered**

$FX$  $XX$

**VSR Data Layout for xvcvsxddp**

<table>
<thead>
<tr>
<th>$src$</th>
<th>$VSR[XB].dword[0]$</th>
<th>$VSR[XB].dword[1]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tgt$</td>
<td>$VSR[XT].dword[0]$</td>
<td>$VSR[XT].dword[1]$</td>
</tr>
</tbody>
</table>
## VSX Vector Convert with round Signed Doubleword to Single-Precision format XX2-form

**xvcvsxdsp**

<table>
<thead>
<tr>
<th><strong>XT, XB</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
</tr>
</tbody>
</table>

- **if MSR.VSX=0 then VSX_Unavailable();**

- **ex_flag ← 0b0**

- **do i = 0 to 1**

  - **reset_xflags();**

  - **src ← bfp_CONVERT_FROM_SI64(VSR[32×BX+B].dword[i]);**

  - **rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v);**

  - **vresult.dword[i].word[0] ← bfp32_CONVERT_FROM_BFP(rnd);**

  - **vresult.dword[i].word[1] ← bfp32_CONVERT_FROM_BFP(rnd);**

  - **if xx_flag=1 then SetFX(FPSCR.XX);**

  - **ex_flag ← ex_flag | (FPSCR.XE & xx_flag);**

- **end**

- **if ex_flag=0 then VSR[32×TX+T] ← vresult**

---

### VSR Data Layout for xvcvsxdsp

<table>
<thead>
<tr>
<th><strong>src</strong></th>
<th>VSR[XB].dword[0]</th>
<th>VSR[XB].dword[1]</th>
</tr>
</thead>
</table>

Let **XT** be the value **32×TX + T**.
Let **XB** be the value **32×BX + B**.

For each integer value **i** from 0 to 1, do the following.
Let **src** be the signed integer in doubleword element **i** of VSR[XB].

**src** is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by **RN**.

The result is placed into bits 0:31 of doubleword element **i** of VSR[XT] in single-precision format.

The result is also placed into bits 32:63 of doubleword element **i** of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

### Special Registers Altered

**FX XX**

#### Programming Note

Previous versions of the architecture allowed the contents of words 1 and 3 of the result register to be undefined. However, all processors that support this instruction write the result into words 0 and 1 and words 2 and 3 of the result register, as is required by this version of the architecture.
**VSX Vector Convert Signed Word to Double-Precision format XX2-form**

\[
\text{xvcvsxwp} \quad \text{XT,XB}
\]

```
if MSR.VSX=0 then VSXUnavailable()

do i = 0 to 1
    src ← bfp\_CONVERT\_FROM\_SI32(VSR[32×BX+B].dword[i].word[0])
    VSR[32×TX+T].dword[i] ← bfp\_ROUND\_TO\_BF64(FPSCR.RN,src)
end
```

Let \(XT\) be the value \(32×TX + T\).
Let \(XB\) be the value \(32×BX + B\).

For each integer value \(i\) from 0 to 1, do the following.

Let \(src\) be the signed integer value in bits 0:31 of doubleword element \(i\) of \(VSR[XB]\).

\(src\) is placed into doubleword element \(i\) of \(VSR[XT]\) in double-precision format.

**Special Registers Altered**

None

**VSR Data Layout for xvcvsxwp**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].word[0]</th>
<th>unused</th>
<th>VSR[XB].word[2]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 32 64 96 127</td>
<td>0 32 64 96 127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VSX Vector Convert with round Signed Word to Single-Precision format XX2-form**

\[
\text{xvcvsxwp} \quad \text{XT,XB}
\]

```
if MSR.VSX=0 then VSXUnavailable()

ex_flag ← 0b0

do i = 0 to 3
    reset\_xflags()
    src ← bfp\_CONVERT\_FROM\_SI32(VSR[32×BX+B].word[i])
    rnd ← bfp\_ROUND\_TO\_BF32(FPSCR.RN,src)
    vresult.word[i] ← bfp\_CONVERT\_FROM\_BF32(FPSCR.XE & xx_flag,end)
    if xx_flag=1 then SetFX(FPSCR.XX)
    ex_flag ← ex_flag | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let \(XT\) be the value \(32×TX + T\).
Let \(XB\) be the value \(32×BX + B\).

For each integer value \(i\) from 0 to 3, do the following.

Let \(src\) be the signed integer in word element \(i\) of \(VSR[XB]\).

\(src\) is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by \(RN\).

The result is placed into word element \(i\) of \(VSR[XT]\) in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to \(VSR[XT]\).

**Special Registers Altered**

\(FX \ XX\)
VSX Vector Convert with round Unsigned Doubleword to Double-Precision format
XX2-form

\[
xvcvuxddp \quad XT,XB
\]

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 1, do the following.
Let \( src \) be the unsigned integer in doubleword element \( i \) of \( VSR[XB] \).

\( src \) is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by \( RN \).

The result is placed into doubleword element \( i \) of \( VSR[XT] \) in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( VSR[XT] \).

Special Registers Altered
FX XX

VSR Data Layout for xvcvuxddp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>VSR[XB].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>
VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form

VSR Data Layout for xvcvuxdsp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>VSR[XB].dword[1]</th>
</tr>
</thead>
</table>

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.
Let src be the unsigned integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by RN.

The result is placed into bits 0:31 of doubleword element i of VSR[XT] in single-precision format.

The result is also placed into bits 32:63 of doubleword element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
FX XX

Programming Note

Previous versions of the architecture allowed the contents of words 1 and 3 of the result register to be undefined. However, all processors that support this instruction write the result into words 0 and 1 and words 2 and 3 of the result register, as is required by this version of the architecture.
VSX Vector Convert Unsigned Word to Double-Precision format XX2-form

\[ \text{xvcvuxwdp} \quad \text{XT, XB} \]

1. If MSR.VSX=0 then VSX_Unavailable()

2. Do \( i = 0 \) to 1
   - \( \text{src} \leftarrow \text{bfp\_CONVERT\_FROM\_UI32(VSR[32\times BX+B].dword[i].word[0])} \)
   - \( \text{VSR[32\times TX+T].dword[i] } \leftarrow \text{bfp64\_CONVERT\_FROM\_BFP(src)} \)
   - End

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 1, do the following.
Let \( \text{src} \) be the unsigned integer value in bits 0:31 of doubleword element \( i \) of \( VSR[XB] \).
\( \text{src} \) is placed into doubleword element \( i \) of \( VSR[XT] \) in double-precision format.

Special Registers Altered
None

VSX Vector Convert with round Unsigned Word to Single-Precision format XX2-form

\[ \text{xvcvuxwsp} \quad \text{XT, XB} \]

1. If MSR.VSX=0 then VSX_Unavailable()

2. \( \text{ex\_flag} \leftarrow 0b0 \)

3. Do \( i = 0 \) to 3
   - \( \text{reset\_xflags()} \)
   - \( \text{src} \leftarrow \text{bfp\_CONVERT\_FROM\_UI32(VSR[32\times BX+B].word[i])} \)
   - \( \text{rnd} \leftarrow \text{bfp\_ROUND\_TO\_BFP32(FPSCR.RN,v)} \)
   - \( \text{vresult.word[i]} \leftarrow \text{bfp32\_CONVERT\_FROM\_BFP(rnd)} \)
   - If \( xx\_flag=1 \) then SetFX(FPSCR.XX)
   - \( \text{ex\_flag} \leftarrow \text{ex\_flag | (FPSCR.XE & xx\_flag)} \)
   - End

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 3, do the following.
Let \( \text{src} \) be the unsigned integer value in word element \( i \) of \( VSR[XB] \).
\( \text{src} \) is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by \( RN \).

The result is placed into word element \( i \) of \( VSR[XT] \) in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( VSR[XT] \).

Special Registers Altered
FX XX
VSX Vector Divide Double-Precision XX3-form

```
xdivdp XT,XA,XB
```

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>120</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>30</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

doi = 0 to 1

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])

src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])

v ← bfp_DIVIDE(src1,src2)

rnd ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)

vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

if vxidi_flag=1 then SetFX(FPSCR.VXIDI)

if vxzdz_flag=1 then SetFX(FPSCR.VXZDZ)

if ox_flag=1 then SetFX(FPSCR.OX)

if ux_flag=1 then SetFX(FPSCR.UX)

if xx_flag=1 then SetFX(FPSCR.XX)

if zx_flag=1 then SetFX(FPSCR.ZX)

ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)

| (FPSCR.VE & vxidi_flag)

| (FPSCR.VE & vxzdz_flag)

| (FPSCR.OE & ox_flag)

| (FPSCR.UE & ux_flag)

| (FPSCR.ZE & xx_flag)

| (FPSCR.XE & zx_flag)

end

if ex_flag=0 then VSR[32×TX+T] ← vresult

Special Registers Altered

FX OX UX ZX XX VXSNAN VXIDI VXZDZ

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is divided[1] by src2, producing a quotient having unbounded range and precision.

The quotient is normalized[2].

See Table 124.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.


The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 114, “Vector Floating-Point Final Result,” on page 823.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX ZX XX VXSNAN VXIDI VXZDZ

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Explanation:

- **src1**: The double-precision floating-point value in doubleword element \( i \) of \( \text{VSR}[X_A] \) (where \( i = \{0, 1\} \)).
- **src2**: The double-precision floating-point value in doubleword element \( j \) of \( \text{VSR}[X_B] \) (where \( j = \{0, 1\} \)).
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- **D(x,y)**: Return the normalized quotient of floating-point value \( x \) divided by floating-point value \( y \), having unbounded range and precision.
- **Q(x)**: Return a QNaN with the payload of \( x \).
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.

### Table 124: Actions for xvdivdp (element i)

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Zero</td>
<td>v ← D(src1, src2)</td>
<td>v ← +Infinity</td>
<td>z_flag ← 1</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>z_flag ← 1</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero</td>
<td>v ← D(src1, src2)</td>
<td>v ← -Infinity</td>
<td>z_flag ← 1</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Zero</td>
<td>v ← D(src1, src2)</td>
<td>v ← -Infinity</td>
<td>z_flag ← 1</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>
VSX Vector Divide Single-Precision XX3-form

\texttt{vxddivsp XT,XA,XB}

\begin{tabular}{cccccc}
  60 & 6 & 11 & 16 & 88 & 133 \\
  0 & 6 & 11 & 16 & 21 & 29 \\
\end{tabular}

\begin{itemize}
  \item if MSR.VSX=0 then \texttt{VSX.Unavailable(\texttt{)})}
  \item ex_flag \leftarrow \texttt{0b0}
  \item do \texttt{i = 0 to 3}
    \begin{itemize}
      \item reset_xflags()
      \item src1 \leftarrow \texttt{bfp_CONVERT_FROM_BFP32(VSR[\texttt{32\times AX + A}].word[i])}
      \item src2 \leftarrow \texttt{bfp_CONVERT_FROM_BFP32(VSR[\texttt{32\times BX + B}].word[i])}
      \item v \leftarrow \texttt{bfp_DIVIDE(src1,src2)}
      \item rnd \leftarrow \texttt{bfp_ROUND_TO_BFP32(FPSCR.RN,v)}
      \item vresult.word[i] \leftarrow \texttt{bfp32_CONVERT_FROM_BFP(rnd)}
    \end{itemize}
  \item if vxsnan_flag=1 then \texttt{SetFX(FPSCR.VXSNAN)}
  \item if vxidi_flag=1 then \texttt{SetFX(FPSCR.VXIDI)}
  \item if vxzsi_flag=1 then \texttt{SetFX(FPSCR.VXZDZ)}
  \item if ox_flag=1 then \texttt{SetFX(FPSCR.OX)}
  \item if ux_flag=1 then \texttt{SetFX(FPSCR.UX)}
  \item if xx_flag=1 then \texttt{SetFX(FPSCR.XX)}
  \item if zx_flag=1 then \texttt{SetFX(FPSCR.ZX)}
  \item ex_flag \leftarrow ex_flag | (FPSCR.VE & vxsnan_flag)
  \item ex_flag \leftarrow ex_flag | (FPSCR.VE & vxidi_flag)
  \item ex_flag \leftarrow ex_flag | (FPSCR.VE & vxzsi_flag)
  \item ex_flag \leftarrow ex_flag | (FPSCR.OE & ox_flag)
  \item ex_flag \leftarrow ex_flag | (FPSCR.UE & ux_flag)
  \item ex_flag \leftarrow ex_flag | (FPSCR.ZE & xx_flag)
  \item ex_flag \leftarrow ex_flag | (FPSCR.XE & zx_flag)
\end{itemize}
\item end
\item if ex_flag=0 then VSR[\texttt{32\times TX + T}] \leftarrow vresult

For each integer value \texttt{i} from 0 to 3, do the following.
Let \texttt{src1} be the single-precision floating-point operand in word element \texttt{i} of \texttt{VSR[\texttt{XA}]}.
Let \texttt{src2} be the single-precision floating-point operand in word element \texttt{i} of \texttt{VSR[\texttt{XB}]}.
\texttt{src1} is divided\textsuperscript{1} by \texttt{src2}, producing a quotient having unbounded range and precision.
The quotient is normalized\textsuperscript{2}.
See Table 125.
The intermediate result is rounded to single-precision using the rounding mode specified by \texttt{RN}.
The result is placed into word element \texttt{i} of \texttt{VSR[\texttt{XT}]} in single-precision format.
See Table 114, “Vector Floating-Point Final Result,” on page 823.
If a trap-enabled exception occurs in any element of the vector, no results are written to \texttt{VSR[\texttt{XT}]}.

Special Registers Altered
\texttt{FX OX UX ZX XX VXSNAN VXIDI VXZDZ}

---

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Zero</td>
<td>v ← D(src1, src2)</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← D(src1, src2)</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Zero</td>
<td>v ← D(src1, src2)</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← D(src1, src2)</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

Explanation:
- **src1** The single-precision floating-point value in word element \( i \) of VSR[XA] (where \( i = \{0, 1, 2, 3\} \)).
- **src2** The single-precision floating-point value in word element \( j \) of VSR[XB] (where \( j = \{0, 1, 2, 3\} \)).
- **dQNaN** Default quiet NaN (0x7FC0_0000).
- **NZF** Nonzero finite number.
- **Rezd** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- **D(x,y)** Return the normalized quotient of floating-point value \( x \) divided by floating-point value \( y \), having unbounded range and precision.
- **Note:** If \( x = -y \), \( v \) is considered to be an exact-zero-difference result (Rezd).
- **Q(x)** Return a QNaN with the payload of \( x \).
- **v** The intermediate result having unbounded significand precision and unbounded exponent range.

Table 125: Actions for xvdivsp (element i)
VSX Vector 16-bit Floating-Point GER (rank-2 update) XX3-form

```
xvf16ger2  AT,XA,XB
```

VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate XX3-form

```
xvf16ger2pp  AT,XA,XB
```

VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate XX3-form

```
xvf16ger2pn  AT,XA,XB
```

VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate XX3-form

```
xvf16ger2np  AT,XA,XB
```

VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate XX3-form

```
xvf16ger2nn  AT,XA,XB
```

Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) MMIRR:XX3-form

```
pxvf16ger2pp  AT,XA,XB,XMSK,YMSK,PMSK
```

Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form

```
pxvf16ger2pp  AT,XA,XB,XMSK,YMSK,PMSK
```

Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form

```
pxvf16ger2pn  AT,XA,XB,XMSK,YMSK,PMSK
```

Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form

```
pxvf16ger2np  AT,XA,XB,XMSK,YMSK,PMSK
```

Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form

```
pxvf16ger2nn  AT,XA,XB,XMSK,YMSK,PMSK
```
if MSR.VSX=0 then VSX_Unavailable()

if "xvf16ger2" | "xvf16ger2pp" | "xvf16ger2pn" | "xvf16ger2np" | "xvf16ger2nn" then do
  PMSK ← 0b1111
  XMSK ← 0b1111
  YMSK ← 0b1111
end

do i = 0 to 3
do j = 0 to 3
  if XMSK.bit[i] & YMSK.bit[j] then do
    reset_flags()
    src10 ← bfp_CONVERT_FROM_BFP16((PMSK.bit[0]=0) ? 0x0000 : VSR[32×AX+A].word[i].hword[0])
    src11 ← bfp_CONVERT_FROM_BFP16((PMSK.bit[1]=0) ? 0x0000 : VSR[32×AX+A].word[i].hword[1])
    src20 ← bfp_CONVERT_FROM_BFP16((PMSK.bit[0]=0) ? 0x0000 : VSR[32×BX+B].word[j].hword[0])
    src21 ← bfp_CONVERT_FROM_BFP16((PMSK.bit[1]=0) ? 0x0000 : VSR[32×BX+B].word[j].hword[1])
    p1 ← bfp_MULTIPLY(src10, src20)
    v1 ← bfp_MULTIPLY_ADD(src11, src21, p1)
    r1 ← bfp_ROUND_TO_BFP32_NO_TRAP(FPSCR.RN, v1)
    if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
    if vxisi_flag=1 then SetFX(FPSCR.VXISI)
    if ox_flag=1 then SetFX(FPSCR.OX)
    if ux_flag=1 then SetFX(FPSCR.UX)
    if xx_flag=1 then SetFX(FPSCR.XX)
    reset_flags()
if "[pm]xvf16ger2" then
  ACC[AT][i].word[j] ← bfp32_CONVERT_FROM_BFP(r1)
else do
  acc ← bfp_CONVERT_FROM_BFP32(ACC[AT][i].word[j])
  if "[pm]xvf16ger2pp" then r2 ← bfp_ADD(r1, acc)
  if "[pm]xvf16ger2pn" then r2 ← bfp_ADD(r1, bfp_NEGATE(acc))
  if "[pm]xvf16ger2np" then r2 ← bfp_ADD(bfp_NEGATE(r1), acc)
  if "[pm]xvf16ger2nn" then r2 ← bfp_ADD(bfp_NEGATE(r1), bfp_NEGATE(acc))
  ACC[AT][i].word[j] ← bfp32_CONVERT_FROM_BFP(r2)
end

Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of single-precision floating-point values.

Let result be a 4×4 matrix of word elements to be used as a temporary Accumulator.
For \texttt{xfv16ger2}, \texttt{xfv16ger2pp}, \texttt{xfv16ger2pn}, \texttt{xfv16ger2np}, or \texttt{xfv16ger2nn}, let \texttt{PMSK=0b11}, \texttt{XMSK=0b1111}, and 
\texttt{YMSK=0b1111}.

For each integer value \(i\) from 0 to 3, and each integer value \(j\) from 0 to 3, do the following.
If bit \(i\) of \texttt{XMSK} is equal to 1 and bit \(j\) of \texttt{YMSK} is equal to 1, do the following.
If bit 0 of \texttt{PMSK} is equal to 1, let \texttt{src10} be the half-precision floating-point value in hword 0 of word element \(i\) of \texttt{VSR[XA]} and let \texttt{src20} be the half-precision floating-point value in hword 0 of word element \(j\) of \texttt{VSR[XB]}. Otherwise, let \texttt{src10} be the value 0.0 and let \texttt{src20} be the value 0.0, causing the product of \texttt{src10} and \texttt{src20} to be 0.0.

If bit 1 of \texttt{PMSK} is equal to 1, let \texttt{src11} be the half-precision floating-point value in hword 1 of word element \(i\) of \texttt{VSR[XA]} and let \texttt{src21} be the half-precision floating-point value in hword 1 of word element \(j\) of \texttt{VSR[XB]}. Otherwise, let \texttt{src11} be the value 0.0 and let \texttt{src21} be the value 0.0, causing the product of \texttt{src11} and \texttt{src21} to be 0.0.

Let \texttt{prod} be the single-precision product of \texttt{src10} and \texttt{src20}.
Let \texttt{msum} be the sum of \texttt{prod} added to the product of \texttt{src11} and \texttt{src21}. \texttt{msum} is rounded to single-precision using the rounding mode specified in \texttt{RN}.

For \texttt{[pm]xfv16ger2}, the rounded \texttt{msum} is placed into word element \(j\) of \texttt{ACC[AT][i]} in single-precision floating-point format.

For \texttt{[pm]xfv16ger2pp}, the rounded \texttt{msum} is added to the single-precision floating-point value in word element \(j\) of \texttt{ACC[AT][i]}, rounded to single-precision using the rounding mode specified in \texttt{RN}, and placed into word element \(j\) of \texttt{ACC[AT][i]} in single-precision floating-point format.

For \texttt{[pm]xfv16ger2pn}, the rounded \texttt{msum} is added to the negation of the single-precision floating-point value in word element \(j\) of \texttt{ACC[AT][i]}, rounded to single-precision using the rounding mode specified in \texttt{RN}, and placed into word element \(j\) of \texttt{ACC[AT][i]} in single-precision floating-point format.

For \texttt{[pm]xfv16ger2np}, the negation of the rounded \texttt{msum} is added to the single-precision floating-point value in word element \(j\) of \texttt{ACC[AT][i]}, rounded to single-precision using the rounding mode specified in \texttt{RN}, and placed into word element \(j\) of \texttt{ACC[AT][i]} in single-precision floating-point format.

For \texttt{[pm]xfv16ger2nn}, the negation of the rounded \texttt{msum} is added to the negation of the single-precision floating-point value in word element \(j\) of \texttt{ACC[AT][i]}, rounded to single-precision using the rounding mode specified in \texttt{RN}, and placed into word element \(j\) of \texttt{ACC[AT][i]} in single-precision floating-point format.

Otherwise, the contents of \texttt{ACC[AT][i][j]} are set to 0x0000_0000.

Unlike other \textit{VSX Vector Floating-Point} instructions, \texttt{ACC[AT]} is always updated by the execution of the instruction, even when a trap-enabled exception occurs. For every multiply-add operation that is performed as part of the execution of this instruction, if an exception occurs as the result of that particular multiply-add operation, the trap-disabled exception result is returned, even if that exception type is trap-enabled. Exception detection is based on the trap-disabled definition. Exception status is accumulated and the appropriate exception status bits in the \texttt{FPSCR} are updated at the completion of execution of the instruction. Otherwise, behavior is the same as any vector floating-point instruction that can cause an exception. Taking a Program interrupt on a trap-enabled exception when interrupts are enabled by \texttt{MSR.PE0} and \texttt{MSR.PE1} is still supported, albeit with the \texttt{ACC[AT]} updated based on a trap-disabled result.

\textbf{Special Registers Altered:}
\begin{verbatim}
FX VXSNAN VXIMZ OK UX XX  (if [pm] xfv16ger2 )
FX VXSNAN VXISI OK UX XX  (if [pm] xfv16ger2pp)
FX VXSNAN VXISI OK UX XX  (if [pm] xfv16ger2pn)
FX VXSNAN VXISI OK UX XX  (if [pm] xfv16ger2np)
FX VXSNAN VXISI OK UX XX  (if [pm] xfv16ger2nn)
\end{verbatim}
Register Operand Data Layout for \([pm]xvf16ger2[pp|pn|np|nn]\)

| VSR\[XA\] | X\[0\][0] | X\[0\][1] | X\[1\][0] | X\[1\][1] | X\[2\][0] | X\[2\][1] | X\[3\][0] | X\[3\][1] |
| VSR\[XB\] | Y\[0\][0] | Y\[1\][0] | Y\[0\][1] | Y\[1\][1] | Y\[0\][2] | Y\[1\][2] | Y\[0\][3] | Y\[1\][3] |
| ACC\[AT\][0] | T\[0\][0] | T\[0\][1] | T\[0\][2] | T\[0\][3] |
| ACC\[AT\][1] | T\[1\][0] | T\[1\][1] | T\[1\][2] | T\[1\][3] |

**Programming Note**

Let \(X\) be the \(4\times2\) matrix of half-precision floating-point values contained in \(VSR[XA]\) in row-major format.
Let \(Y\) be the \(4\times2\) matrix of half-precision floating-point values contained in \(VSR[XB]\) in row-major format.
Let \(ACC[AT]\) be the Accumulator containing a \(4\times4\) matrix of single-precision floating-point values.

Note that floating-point arithmetic is not associative. That is, \((X+Y)+T\) can return a different result than \(X+(Y+T)\).
The ordering specified by the instruction description for any result element \(i,j\) is the order the operations will be performed in hardware. The floating-point operations to implement each result element for \([pm]xvf16ger2[pp|pn|np|nn]\) are shown below.

\([pm]xvf16ger2\) performs the following form of accumulation of two outer products (rank 2 update).

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 3: \\
\quad \text{ACC}[AT][i][j] = \text{fmadds}(X[i][1], Y[j][1], \text{fmuls}(X[i][0], Y[j][0]))
\]

\([pm]xvf16ger2pp\) performs the following form of accumulation of two outer products (rank 2 update).

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 3: \\
\quad \text{ACC}[AT][i][j] = \text{fadds}(\text{fmadds}(X[i][1], Y[j][1], \text{fmuls}(X[i][0], Y[j][0])), \text{ACC}[AT][i][j])
\]

\([pm]xvf16ger2pn\) performs the following form of accumulation of two outer products (rank 2 update).

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 3: \\
\quad \text{ACC}[AT][i][j] = \text{fsubs}(\text{fmadds}(X[i][1], Y[j][1], \text{fmuls}(X[i][0], Y[j][0])), \text{ACC}[AT][i][j])
\]

\([pm]xvf16ger2np\) performs the following form of accumulation of two outer products (rank 2 update).

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 3: \\
\quad \text{ACC}[AT][i][j] = \text{fadds}(\text{fnmadds}(X[i][1], Y[j][1], \text{fmuls}(X[i][0], Y[j][0])), \text{ACC}[AT][i][j])
\]

\([pm]xvf16ger2nn\) performs the following form of accumulation of two outer products (rank 2 update).

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 3: \\
\quad \text{ACC}[AT][i][j] = \text{fsubs}(\text{fnmadds}(X[i][1], Y[j][1], \text{fmuls}(X[i][0], Y[j][0])), \text{ACC}[AT][i][j])
\]
**VSX Vector 32-bit Floating-Point GER (rank-1 update) XX3-form**

```
xvf32ger       AT,XA,XB

|   59 | AT // A B 27 |
|      | 6  9 11 16 21 |
```

**VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form**

```
xvf32gerpp    AT,XA,XB

|   59 | AT // A B 26 |
|      | 6  9 11 16 21 |
```

**VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form**

```
xvf32gerpn    AT,XA,XB

|   59 | AT // A B 154 |
|      | 6  9 11 16 21 |
```

**VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form**

```
xvf32gemp     AT,XA,XB

|   59 | AT // A B 90 |
|      | 6  9 11 16 21 |
```

**VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form**

```
xvf32genn     AT,XA,XB

|   59 | AT // A B 218 |
|      | 6  9 11 16 21 |
```

**Prefix: Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form**

**Positive multiply, Positive accumulate**

```
pxvf32gerpp    AT,XA,XB,XMSK,YMSK

Prefix:

|   1 | 3 9 // // // |
|    | 6 8 12 | 24 | 28 | 31 |
```

**Suffix:

|   59 | AT // A B 26 |
|      | 6  9 11 16 21 |
```

**Prefix: Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form**

```
pxvf32gerpn    AT,XA,XB,XMSK,YMSK

Prefix:

|   1 | 3 9 // // // |
|    | 6 8 12 | 24 | 28 | 31 |
```

**Suffix:

|   59 | AT // A B 154 |
|      | 6  9 11 16 21 |
```

**Prefix: Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form**

```
pxvf32gemp    AT,XA,XB,XMSK,YMSK

Prefix:

|   1 | 3 9 // // // |
|    | 6 8 12 | 24 | 28 | 31 |
```

**Suffix:

|   59 | AT // A B 90 |
|      | 6  9 11 16 21 |
```

**Prefix: Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form**

```
pxvf32genn    AT,XA,XB,XMSK,YMSK

Prefix:

|   1 | 3 9 // // // |
|    | 6 8 12 | 24 | 28 | 31 |
```

**Suffix:

|   59 | AT // A B 218 |
|      | 6  9 11 16 21 |
if MSR.VSX=0 then VSX_Unavailable()

if "xvf32ger" | "xvf32gerpp" | "xvf32gerpn" | "xvf32gernp" | "xvf32gernn" then do
    XMSK ← 0b1111
    YMSK ← 0b1111
end

do i = 0 to 3
    do j = 0 to 3
        if XMSK.bit[i]=1 & YMSK.bit[j]=1 then do
            reset_flags()
            src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
            src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[j])
            acc ← bfp_CONVERT_FROM_BFP32(ACC[AT][i].word[j])
            if "[pm]xvf32ger" then v ← bfp_MULTIPLY( src1, src2 )
            if "[pm]xvf32gerpp" then v ← bfp_MULTIPLY_ADD( src1, src2, acc )
            if "[pm]xvf32gerpn" then v ← bfp_MULTIPLY_ADD( src1, src2, bfp_NEGATE(acc) )
            if "[pm]xvf32gernp" then v ← bfp_MULTIPLY_ADD( src1, src2, bfp_NEGATE(acc) )
            if "[pm]xvf32gernn" then v ← bfp_MULTIPLY_ADD( src1, src2, acc )
        end
        r ← bfp_ROUND_TO_BFP32_NO_TRAP(v)
        if "[pm]xvf32ger" then r ← bfp_NEGATE(r)
        if "[pm]xvf32gerpp" then r ← bfp_NEGATE(r)
        ACC[AT][i].word[j] ← bfp32_CONVERT_FROM_BFP(r)
        if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
        if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
        if vxisi_flag=1 then SetFX(FPSCR.VXISI)
        if ox_flag=1 then SetFX(FPSCR.OX)
        if ux_flag=1 then SetFX(FPSCR.UX)
        if xx_flag=1 then SetFX(FPSCR.XX)
    end
    else
        ACC[AT][i].word[j] ← 0x0000_0000
    end
end

Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of single-precision floating-point values.

For xvf32ger, xvf32gerpp, xvf32gerpn, xvf32gernp, or xvf32gernn, let XMSK=0b1111 and YMSK=0b1111.

For each integer value i from 0 to 3, and each integer value j from 0 to 3, do the following.
    If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.
        Let prod be the product of the single-precision floating-point value in word element i of VSR[XA] and the single-precision floating-point value in word element j of VSR[XB], having unbounded range and precision.
        For [pm]xvf32ger, prod is rounded to single-precision using the rounding mode specified in RN. The rounded result is placed into word element j of ACC[AT][i] in single-precision floating-point format.
        For [pm]xvf32gerpp, the single-precision floating-point value in word element j of ACC[AT][i] is added to prod. The intermediate result is rounded to single-precision using the rounding mode specified in RN. The rounded result is placed into word element j of ACC[AT][i] in single-precision floating-point format.
        For [pm]xvf32gerpn, the single-precision floating-point value in word element j of ACC[AT][i] is subtracted from prod. The intermediate result is rounded to single-precision using the rounding mode specified in RN. The rounded result is placed into word element j of ACC[AT][i] in single-precision floating-point format.
For \([pm]\times vf32gernp\), the single-precision floating-point value in word element \(j\) of ACC[AT][i] is subtracted from prod. The intermediate result is rounded to single-precision using the rounding mode specified in RN. The rounded result is negated and placed into word element \(j\) of ACC[AT][i] in single-precision floating-point format.

For \([pm]\times vf32gernn\), the single-precision floating-point value in word element \(j\) of ACC[AT][i] is added to prod. The intermediate result is rounded to single-precision using the rounding mode specified in RN. The rounded result is negated and placed into word element \(j\) of ACC[AT][i] in single-precision floating-point format.

Otherwise, the contents of word element \(j\) of ACC[AT][i] are set to 0x0000_0000.

Unlike other vector floating-point instructions, ACC[AT] is always updated by the execution of the instruction, even when a trap-enabled exception occurs. For every multiply-add operation that is performed as part of the execution of this instruction, if an exception occurs as the result of that particular multiply-add operation, the trap-disabled exception result is returned, even if that exception type is trap-enabled. Exception detection is based on the trap-disabled definition. Exception status is and the appropriate exception status bits in the FPSCR are updated at the completion of execution of the instruction. Otherwise, behavior is the same as any vector floating-point instruction that can cause an accumulated exception. Taking a Program interrupt on a trap-enabled exception when interrupts are enabled by MSR.FE0 and MSR.FE1 is still supported, albeit with the ACC[AT] updated based on a trap-disabled result.

**Special Registers Altered:**

- FX VXSNAN VXIMZ OX UX XX (if \([pm]\times vf32ger\))
- FX VXSNAN VXISI OX UX XX (if \([pm]\times vf32gernpp\))
- FX VXSNAN VXISI OX UX XX (if \([pm]\times vf32gernpn\))
- FX VXSNAN VXISI OX UX XX (if \([pm]\times vf32gernp\))
- FX VXSNAN VXISI OX UX XX (if \([pm]\times vf32gernn\))

**Register Operand Data Layout for \([pm]\times vf32ger[pp|pn|np|nn]\)**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC[AT][0]</td>
<td>T[0][0]</td>
<td>T[0][1]</td>
<td>T[0][2]</td>
<td>T[0][3]</td>
</tr>
</tbody>
</table>
Let $X$ be the 4-element vector of single-precision floating-point values contained in $VSR[XA]$. Let $Y$ be the 4-element vector of single-precision floating-point values contained in $VSR[XB]$. Let $ACC[AT]$ be the Accumulator containing a $4\times 4$ matrix of single-precision floating-point values.

The floating-point operations to implement each result element for $xvf32ger|pp|pn|nn$ are shown below.

$[pm]xvf32ger$ performs the following form of accumulation of one outer product (rank 1 update).

$$\text{for } i=0 \text{ to } 3, \ j=0 \text{ to } 3:\ \ ACC[AT][i][j] = \text{fmuls}(X[i],Y[j])$$

$[pm]xvf32gerpp$ performs the following form of accumulation of one outer product (rank 1 update).

$$\text{for } i=0 \text{ to } 3, \ j=0 \text{ to } 3:\ \ ACC[AT][i][j] = \text{fadd}(X[i],Y[j],ACC[AT][i][j])$$

$[pm]xvf32gerpn$ performs the following form of accumulation of one outer product (rank 1 update).

$$\text{for } i=0 \text{ to } 3, \ j=0 \text{ to } 3:\ \ ACC[AT][i][j] = \text{fsub}(X[i],Y[j],ACC[AT][i][j])$$

$[pm]xvf32gernp$ performs the following form of accumulation of one outer product (rank 1 update).

$$\text{for } i=0 \text{ to } 3, \ j=0 \text{ to } 3:\ \ ACC[AT][i][j] = \text{fnmsub}(X[i],Y[j],ACC[AT][i][j])$$

$[pm]xvf32gernn$ performs the following form of accumulation of one outer product (rank 1 update).

$$\text{for } i=0 \text{ to } 3, \ j=0 \text{ to } 3:\ \ ACC[AT][i][j] = \text{fnmadd}(X[i],Y[j],ACC[AT][i][j])$$
VSX Vector 64-bit Floating-Point GER (rank-1 update) XX3-form

\[ \text{xvf64ger} \ AT, XAp, XB \]

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form

\[ \text{pmxvf64gerpp} \ AT, XAp, XB, XMSK, YMSK \]

Prefix:

\[ \begin{array}{ccccccc}
\text{AT} & \text{Ap} & \text{B} & \text{XMSK} & \text{YMSK} \\
0 & 6 & 9 & 12 & 14 & 15 & 16 & 24 & 28 & 31
\end{array} \]

Suffix:

\[ \begin{array}{ccccccc}
\text{AT} & \text{Ap} & \text{B} & \text{XMSK} & \text{YMSK} \\
0 & 6 & 9 & 12 & 14 & 15 & 16 & 24 & 28 & 31
\end{array} \]

VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form

\[ \text{xvf64gerpp} \ AT, XAp, XB \]

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form

\[ \text{pmxvf64gerpp} \ AT, XAp, XB, XMSK, YMSK \]

Prefix:

\[ \begin{array}{ccccccc}
\text{AT} & \text{Ap} & \text{B} & \text{XMSK} & \text{YMSK} \\
0 & 6 & 9 & 12 & 14 & 15 & 16 & 24 & 28 & 31
\end{array} \]

Suffix:

\[ \begin{array}{ccccccc}
\text{AT} & \text{Ap} & \text{B} & \text{XMSK} & \text{YMSK} \\
0 & 6 & 9 & 12 & 14 & 15 & 16 & 24 & 28 & 31
\end{array} \]

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form

\[ \text{xvf64gerpn} \ AT, XAp, XB \]

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form

\[ \text{pmxvf64gerpn} \ AT, XAp, XB, XMSK, YMSK \]

Prefix:

\[ \begin{array}{ccccccc}
\text{AT} & \text{Ap} & \text{B} & \text{XMSK} & \text{YMSK} \\
0 & 6 & 9 & 12 & 14 & 15 & 16 & 24 & 28 & 31
\end{array} \]

Suffix:

\[ \begin{array}{ccccccc}
\text{AT} & \text{Ap} & \text{B} & \text{XMSK} & \text{YMSK} \\
0 & 6 & 9 & 12 & 14 & 15 & 16 & 24 & 28 & 31
\end{array} \]

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form

\[ \text{xvf64gemp} \ AT, XAp, XB \]

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form

\[ \text{pmxvf64gemp} \ AT, XAp, XB, XMSK, YMSK \]

Prefix:

\[ \begin{array}{ccccccc}
\text{AT} & \text{Ap} & \text{B} & \text{XMSK} & \text{YMSK} \\
0 & 6 & 9 & 12 & 14 & 15 & 16 & 24 & 28 & 31
\end{array} \]

Suffix:

\[ \begin{array}{ccccccc}
\text{AT} & \text{Ap} & \text{B} & \text{XMSK} & \text{YMSK} \\
0 & 6 & 9 & 12 & 14 & 15 & 16 & 24 & 28 & 31
\end{array} \]

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form

\[ \text{xvf64genn} \ AT, XAp, XB \]

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form

\[ \text{pmxvf64genn} \ AT, XAp, XB, XMSK, YMSK \]

Prefix:

\[ \begin{array}{ccccccc}
\text{AT} & \text{Ap} & \text{B} & \text{XMSK} & \text{YMSK} \\
0 & 6 & 9 & 12 & 14 & 15 & 16 & 24 & 28 & 31
\end{array} \]

Suffix:

\[ \begin{array}{ccccccc}
\text{AT} & \text{Ap} & \text{B} & \text{XMSK} & \text{YMSK} \\
0 & 6 & 9 & 12 & 14 & 15 & 16 & 24 & 28 & 31
\end{array} \]
if MSR.VSX=0 then VSX_Unavailable()

if "xvf64ger" | "xvf64gerpp" | "xvf64gerpn" | "xvf64gernp" | "xvf64gernn" then do
    XMSK ← 0b1111
    YMSK ← 0b11
end

vsr1.qword[0] ← VSR[32×AX+Ap]
vsr2 ← VSR[32×BX+B]
doi.i = 0 to 3
do j = 0 to 1
    if XMSK.bit[i]=1 & YMSK.bit[j]=1 then do
        reset_flags()
        src1 ← bfp_CONVERT_FROM_BFP64(vsr1.dword[i])
        src2 ← bfp_CONVERT_FROM_BFP64(vsr1.dword[j])
        acc ← bfp_CONVERT_FROM_BFP64(ACC[AT][i].dword[j])
        if "[pm]xvf64ger" then v ← bfp_MULTIPLY( src1, src2 )
        if "[pm]xvf64gerpp" then v ← bfp_MULTIPLY_ADD( src1, src2, acc )
        if "[pm]xvf64gerpn" then v ← bfp_MULTIPLY_ADD( src1, src2, bfp_NEGATE(acc) )
        if "[pm]xvf64gernp" then v ← bfp_MULTIPLY_ADD( src1, src2, bfp_NEGATE(acc) )
        if "[pm]xvf64gernn" then v ← bfp_MULTIPLY_ADD( src1, src2, acc )
    end
    r ← bfp_ROUND_TO_BFP64_NO_TRAP(v)
    if "[pm]xvf64ger" then r ← bfp_NEGATE(r)
    if "[pm]xvf64gerpp" then r ← bfp_NEGATE(r)
    ACC[AT][i].dword[j] ← bfp64_CONVERT_FROM_BFP(r)
    if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
    if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
    if vxisi_flag=1 then SetFX(FPSCR.VXISI)
    if ox_flag=1 then SetFX(FPSCR.OX)
    if ux_flag=1 then SetFX(FPSCR.UX)
    if xx_flag=1 then SetFX(FPSCR.XX)
end
else
    ACC[AT][i].dword[j] ← 0x0000_0000_0000_0000
end

Let XAp be the value of 32×AX + Ap. If XAp is odd, or is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×2 matrix of double-precision floating-point values.
Let vsrcX be the concatenation of the contents of VSR[XAp] and VSR[XAp+1].
Let vsrCY be the contents of VSR[XB].

For xvf64ger, xvf64gerpp, xvf64gerpn, xvf64gernp, and xvf64gernn, let XMSK=0b1111 and YMSK=0b11.

For each integer value i from 0 to 3, and each integer value j from 0 to 1, do the following.

If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.

Let prod be the product of the double-precision floating-point value in doubleword element i of vsrcX and the double-precision floating-point value in doubleword element j of vsrCY, having unbounded range and precision.

For [pm]xvf64ger, prod is rounded to double-precision using the rounding mode specified in RN. The rounded result is placed into doubleword element j of ACC[AT][i] in double-precision floating-point format.
For \([pm]xvf64gerpp\), \(prod\) is added to the double-precision floating-point value in doubleword element \(j\) of \(ACC[AT][i]\). The intermediate result is rounded to double-precision using the rounding mode specified in \(RN\). The rounded result is placed into doubleword element \(j\) of \(ACC[AT][i]\) in double-precision floating-point format.

For \([pm]xvf64ger\), \(prod\) is added to the negation of the double-precision floating-point value in doubleword element \(j\) of \(ACC[AT][i]\). The intermediate result is rounded to double-precision using the rounding mode specified in \(RN\). The rounded result is placed into doubleword element \(j\) of \(ACC[AT][i]\) in double-precision floating-point format.

For \([pm]xvf64gernn\), \(prod\) is added to the double-precision floating-point value in doubleword element \(j\) of \(ACC[AT][i]\). The intermediate result is rounded to double-precision using the rounding mode specified in \(RN\). The rounded result is negated and placed into doubleword element \(j\) of \(ACC[AT][i]\) in double-precision floating-point format.

For \([pm]xvf64gernp\), \(prod\) is added to the negation of the double-precision floating-point value in doubleword element \(j\) of \(ACC[AT][i]\). The intermediate result is rounded to double-precision using the rounding mode specified in \(RN\). The rounded result is negated and placed into doubleword element \(j\) of \(ACC[AT][i]\) in double-precision floating-point format.

Otherwise, the contents of doubleword element \(j\) of \(ACC[AT][i]\) are set to 0x0000_0000_0000_0000.

Unlike other vector floating-point instructions, \(ACC[AT]\) is always updated by the execution of the instruction, even when a trap-enabled exception occurs. For every multiply-add operation that is performed as part of the execution of this instruction, if an exception occurs as the result of that particular multiply-add operation, the trap-disabled exception result is returned, even if that exception type is trap-enabled. Exception detection is based on the trap-disable definition. Exception status is accumulated and the appropriate exception status bits in the FPSCR are updated at the completion of execution of the instruction. Otherwise, behavior is the same as any vector floating-point instruction that can cause an exception. Taking a Program interrupt on a trap-enabled exception when interrupts are enabled by MSR.FE0 and MSR.FE1 is still supported, albeit with the \(ACC[AT]\) updated based on a trap-disabled result.

Special Registers Altered:

<table>
<thead>
<tr>
<th>Special Registers Altered:</th>
<th>FX VXSNAN VXIMZ OX UX XX</th>
<th>FX VXSNAN VXIMZ VXISI OX UX XX</th>
<th>FX VXSNAN VXIMZ VXISI OX UX XX</th>
<th>FX VXSNAN VXIMZ VXISI OX UX XX</th>
<th>FX VXSNAN VXIMZ VXISI OX UX XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX VXSNAN VXIMZ VXISI OX UX XX</td>
<td>(if ([pm]xvf64ger))</td>
<td>(if ([pm]xvf64gerpp))</td>
<td>(if ([pm]xvf64gernp))</td>
<td>(if ([pm]xvf64gern))</td>
<td>(if ([pm]xvf64gernn))</td>
</tr>
</tbody>
</table>

Register Operand Data Layout for \([pm]xvf64ger[pp|pn|np|nn]\):

<table>
<thead>
<tr>
<th>VSR[XAp]</th>
<th>X[0]</th>
<th>X[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[XB]</td>
<td>Y[0]</td>
<td>Y[1]</td>
</tr>
<tr>
<td>ACC[AT][0]</td>
<td>T[0][0]</td>
<td>T[0][1]</td>
</tr>
<tr>
<td>ACC[AT][1]</td>
<td>T[1][0]</td>
<td>T[1][1]</td>
</tr>
<tr>
<td>ACC[AT][3]</td>
<td>T[3][0]</td>
<td>T[3][1]</td>
</tr>
</tbody>
</table>
Let \( X \) be the 4-element vector of double-precision floating-point values contained in the concatenation of \( \text{VSR}[XAp] \) and \( \text{VSR}[XAp+1] \).
Let \( Y \) be the 2-element vector of double-precision floating-point values contained in \( \text{VSR}[XB] \).
Let \( \text{ACC}[AT] \) be the Accumulator containing a \( 4 \times 2 \) matrix of double-precision floating-point values.

The floating-point operations to implement each result element for \( \text{xvf64ger} [pp|pn|np|nn] \) are shown below.

\[
\text{pm} \text{xvf64ger} \text{ performs the following form of accumulation of one outer product (rank 1 update).}
\]

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 1:
\quad \text{ACC}[AT][i][j] = \text{fmul}(X[i], Y[j])
\]

\[
\text{pm} \text{xvf64gerpp} \text{ performs the following form of accumulation of one outer product (rank 1 update).}
\]

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 1:
\quad \text{ACC}[AT][i][j] = \text{fmadd}(X[i], Y[j], \text{ACC}[AT][i][j])
\]

\[
\text{pm} \text{xvf64gerpn} \text{ performs the following form of accumulation of one outer product (rank 1 update).}
\]

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 1:
\quad \text{ACC}[AT][i][j] = \text{fmsub}(X[i], Y[j], \text{ACC}[AT][i][j])
\]

\[
\text{pm} \text{xvf64gernp} \text{ performs the following form of accumulation of one outer product (rank 1 update).}
\]

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 1:
\quad \text{ACC}[AT][i][j] = \text{fnmsub}(X[i], Y[j], \text{ACC}[AT][i][j])
\]

\[
\text{pm} \text{xvf64gernn} \text{ performs the following form of accumulation of one outer product (rank 1 update).}
\]

\[
\text{for } i=0 \text{ to } 3, j=0 \text{ to } 1:
\quad \text{ACC}[AT][i][j] = \text{fnmadd}(X[i], Y[j], \text{ACC}[AT][i][j])
\]
VSX Vector 4-bit Signed Integer GER (rank-8 update) XX3-form

\[
xvi4ger8 \quad AT,XA,XB
\]

VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate XX3-form

\[
xvi4ger8pp \quad AT,XA,XB
\]

Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) MMIRR:XX3-form

\[
\text{pmxvi4ger8} \quad AT,XA,XB,XMSK,YMSK,PMSK
\]

Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate MMIRR:XX3-form

\[
\text{pmxvi4ger8pp} \quad AT,XA,XB,XMSK,YMSK,PMSK
\]
Let $X_A$ be the value of $32 \times X_A + A$. If $X_A$ is in the range $4 \times X_A$ to $4 \times X_A + 3$, the instruction form is invalid. Let $X_B$ be the value of $32 \times X_B + B$. If $X_B$ is in the range $4 \times X_B$ to $4 \times X_B + 3$, the instruction form is invalid.

Let the contents of $ACC[AT][i].word[j]$ be a $4 \times 4$ matrix of 32-bit signed integer values.

For `xvi4ger8` or `xvi4ger8pp`, let $PMSK=0b11111111$, $XMSK=0b1111$, and $YMSK=0b1111$.

For each integer value $i$ from 0 to 3, and each integer value $j$ from 0 to 3, do the following.

If bit $i$ of $XMSK$ is equal to 1 and bit $j$ of $YMSK$ is equal to 1, do the following.

If bit 0 of $PMSK$ is equal to 1, let prod0 be the product of the 4-bit signed integer value in nibble 0 of word element $i$ of $VSR[X_A] (K[i][0])$ and the 4-bit signed integer value in byte 0 of word element $j$ of $VSR[X_B] (Y[j][0])$, sign-extended to 32 bits. Otherwise, let prod0 be the value 0.

If bit 1 of $PMSK$ is equal to 1, let prod1 be the product of the 4-bit signed integer value in nibble 1 of word element $i$ of $VSR[X_A] (K[i][1])$ and the 4-bit signed integer value in nibble 1 of word element $j$ of $VSR[X_B] (Y[j][1])$, sign-extended to 32 bits. Otherwise, let prod1 be the value 0.

If bit 2 of $PMSK$ is equal to 1, let prod2 be the product of the 4-bit signed integer value in nibble 2 of word element $i$ of $VSR[X_A] (K[i][2])$ and the 4-bit signed integer value in nibble 2 of word element $j$ of $VSR[X_B] (Y[j][2])$, sign-extended to 32 bits. Otherwise, let prod2 be the value 0.

If bit 3 of $PMSK$ is equal to 1, let prod3 be the product of the 4-bit signed integer value in nibble 3 of word element $i$ of $VSR[X_A] (K[i][3])$ and the 4-bit signed integer value in nibble 3 of word element $j$ of $VSR[X_B] (Y[j][3])$, sign-extended to 32 bits. Otherwise, let prod3 be the value 0.

If bit 4 of $PMSK$ is equal to 1, let prod4 be the product of the 4-bit signed integer value in nibble 4 of word element $i$ of $VSR[X_A] (K[i][4])$ and the 4-bit signed integer value in nibble 4 of word element $j$ of $VSR[X_B] (Y[j][4])$, sign-extended to 32 bits. Otherwise, let prod4 be the value 0.

If bit 5 of $PMSK$ is equal to 1, let prod5 be the product of the 4-bit signed integer value in nibble 5 of word element $i$ of $VSR[X_A] (K[i][5])$ and the 4-bit signed integer value in nibble 5 of word element $j$ of $VSR[X_B] (Y[j][5])$, sign-extended to 32 bits. Otherwise, let prod5 be the value 0.

If bit 6 of $PMSK$ is equal to 1, let prod6 be the product of the 4-bit signed integer value in nibble 6 of word element $i$ of $VSR[X_A] (K[i][6])$ and the 4-bit signed integer value in nibble 6 of word element $j$ of $VSR[X_B] (Y[j][6])$, sign-extended to 32 bits. Otherwise, let prod6 be the value 0.

If bit 7 of $PMSK$ is equal to 1, let prod7 be the product of the 4-bit signed integer value in nibble 7 of word element $i$ of $VSR[X_A] (K[i][7])$ and the 4-bit signed integer value in nibble 7 of word element $j$ of $VSR[X_B] (Y[j][7])$, sign-extended to 32 bits. Otherwise, let prod7 be the value 0.

Let $psum$ be the sum of prod0, prod1, prod2, prod3, prod4, prod5, prod6, and prod7.

For `[pm]xvi4ger8`, $psum$ is placed into word element $j$ of $ACC[AT][i]$ in 32-bit signed integer format.

For `[pm]xvi4ger8ppp`, $psum$ is added to the 32-bit signed integer value in word element $j$ of $ACC[AT][i]$, and the result is placed into word element $j$ of $ACC[AT][i]$ in 32-bit signed integer format.

Otherwise, let $ACC[AT][i][j]$ is set to $0x0000_0000$.

Special Registers Altered:

None
Let $X$ be the $8\times4$ matrix of 4-bit signed integer values contained in $VSR[XA]$ in row-major format.
Let $Y$ be the $8\times4$ matrix of 4-bit signed integer values contained in $VSR[XB]$ in row-major format.
Let $ACC[AT]$ be the Accumulator containing a $4\times4$ matrix of 32-bit signed-integer values.

$[pm]xvi4ger8$ performs the following form of accumulation of eight outer products (rank 8 update).

$$
ACC[AT][i][j] = \text{si32\_CHOP}( EXTS(X[i][0]) \times EXTS(Y[j][0]) + EXTS(X[i][1]) \times EXTS(Y[j][1]) + EXTS(X[i][2]) \times EXTS(Y[j][2]) + EXTS(X[i][3]) \times EXTS(Y[j][3]) + EXTS(X[i][4]) \times EXTS(Y[j][4]) + EXTS(X[i][5]) \times EXTS(Y[j][5]) + EXTS(X[i][6]) \times EXTS(Y[j][6]) + EXTS(X[i][7]) \times EXTS(Y[j][7]))
$$

$[pm]xvi4ger8pp$ performs the following form of accumulation of eight outer products (rank 8 update).

$$
ACC[AT][i][j] = \text{si32\_CHOP}( EXTS(X[i][0]) \times EXTS(Y[j][0]) + EXTS(X[i][1]) \times EXTS(Y[j][1]) + EXTS(X[i][2]) \times EXTS(Y[j][2]) + EXTS(X[i][3]) \times EXTS(Y[j][3]) + EXTS(X[i][4]) \times EXTS(Y[j][4]) + EXTS(X[i][5]) \times EXTS(Y[j][5]) + EXTS(X[i][6]) \times EXTS(Y[j][6]) + EXTS(X[i][7]) \times EXTS(Y[j][7]) + EXTS(ACC[AT][i][j]))
$$
VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) XX3-form

xvi8ger4 AT,XA,XB

if MSR.VSX=0 then VSX_Unavailable()
if "xvi8ger4" then do
  PMSK ← 0b1111
  XMSK ← 0b1111
  YMSK ← 0b1111
end

for i = 0 to 3
  for j = 0 to 3
    if XMSK.bit[i] & YMSK.bit[j] then do
      prod0 ← PMSK.bit[0]=0 ? 0 : EXTS(VSR[32×AX+A].word[i].byte[0]) * EXTZ(VSR[32×BX+B].word[j].byte[0])
      prod1 ← PMSK.bit[1]=0 ? 0 : EXTS(VSR[32×AX+A].word[i].byte[1]) * EXTZ(VSR[32×BX+B].word[j].byte[1])
      psum ← prod0 + prod1 + prod2 + prod3
      if "[pm]xvi8ger4" then ACC[AT][i].word[j] ← CHOP32( psum )
      if "[pm]xvi8ger4pp" then ACC[AT][i].word[j] ← CHOP32( psum + EXTS(ACC[AT][i].word[j]) )
    end
  end
else
  ACC[AT][i][j] ← 0x0000_0000
end
Let \( X_A \) be the value of \( 32 \times A + A \). If \( X_A \) is in the range \( 4 \times AT \) to \( 4 \times AT+3 \), the instruction form is invalid.

Let \( X_B \) be the value of \( 32 \times B + B \). If \( X_B \) is in the range \( 4 \times AT \) to \( 4 \times AT+3 \), the instruction form is invalid.

Let the contents of \( ACC[AT] \) be a \( 4 \times 4 \) matrix of 32-bit signed integer values.

For \textit{xvi8ger4} or \textit{xvi8ger4pp}, let \( PMSK=0b1111 \), \( XMSK=0b1111 \), and \( YMSK=0b1111 \).

For each integer value \( i \) from 0 to 3, and each integer value \( j \) from 0 to 3, do the following.

If bit 0 of \( XMSK \) is equal to 1 and bit \( j \) of \( YMSK \) is equal to 1, do the following.

If bit 0 of \( PMSK \) is equal to 1, let \( prod_0 \) be the product of the 8-bit signed integer value in byte 0 of word element \( i \) of \( VSR[XA] \) \((X[i][0])\) and the 8-bit unsigned integer value in byte 0 of word element \( j \) of \( VSR[XB] \) \((Y[j][0])\), sign-extended to 32 bits. Otherwise, let \( prod_0 \) be the value 0.

If bit 1 of \( PMSK \) is equal to 1, let \( prod_1 \) be the product of the 8-bit signed integer value in byte 1 of word element \( i \) of \( VSR[XA] \) \((X[i][1])\) and the 8-bit unsigned integer value in byte 1 of word element \( j \) of \( VSR[XB] \) \((Y[j][1])\), sign-extended to 32 bits. Otherwise, let \( prod_1 \) be the value 0.

If bit 2 of \( PMSK \) is equal to 1, let \( prod_2 \) be the product of the 8-bit signed integer value in byte 2 of word element \( i \) of \( VSR[XA] \) \((X[i][2])\) and the 8-bit unsigned integer value in byte 2 of word element \( j \) of \( VSR[XB] \) \((Y[j][2])\), sign-extended to 32 bits. Otherwise, let \( prod_2 \) be the value 0.

If bit 3 of \( PMSK \) is equal to 1, let \( prod_3 \) be the product of the 8-bit signed integer value in byte 3 of word element \( i \) of \( VSR[XA] \) \((X[i][3])\) and the 8-bit unsigned integer value in byte 3 of word element \( j \) of \( VSR[XB] \) \((Y[j][3])\), sign-extended to 32 bits. Otherwise, let \( prod_3 \) be the value 0.

Let \( sum \) be the sum of \( prod_0, prod_1, prod_2, \) and \( prod_3 \).

For \textit{pm\(xvi8ger4\)}, \( sum \) is placed into word element \( j \) of \( ACC[AT][i] \) in 32-bit signed integer format.

For \textit{pm\(xvi8ger4pp\)}, \( sum \) is added to the 32-bit signed integer value in word element \( j \) of \( ACC[AT][i] \), and the result is placed into word element \( j \) of \( ACC[AT][i] \) in 32-bit signed integer format.

Otherwise, let word element \( j \) of \( ACC[AT][i] \) is set to \( 0x0000_0000 \).

Special Registers Altered:

None

<table>
<thead>
<tr>
<th>Register Operand Data Layout for \textit{pm(xvi8ger4(pp)}]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ACC[AT][0] )</td>
</tr>
<tr>
<td>( ACC[AT][1] )</td>
</tr>
</tbody>
</table>

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 127
Let $X$ be the $4 \times 4$ matrix of 8-bit signed integer values contained in VSR[XA] in row-major format. Let $Y$ be the $4 \times 4$ matrix of 8-bit unsigned integer values contained in VSR[XB] in row-major format. Let $ACC[AT]$ be the Accumulator containing a $4 \times 4$ matrix of 32-bit signed-integer values.

**Programming Note**

(pm)xvi8ger4 performs the following form of accumulation of four outer products (rank 4 update).

\[
 ACC[AT][i][j] = \text{si32\_CHOP}( \text{EXTS}(X[i][0]) \times \text{EXTZ}(Y[j][0]) + \text{EXTS}(X[i][1]) \times \text{EXTZ}(Y[j][1]) + \text{EXTS}(X[i][2]) \times \text{EXTZ}(Y[j][2]) + \text{EXTS}(X[i][3]) \times \text{EXTZ}(Y[j][3]) )
\]

(pm)xvi8ger4pp performs the following form of accumulation of four outer products (rank 4 update).

\[
 ACC[AT][i][j] = \text{si32\_CHOP}( \text{EXTS}(X[i][0]) \times \text{EXTZ}(Y[j][0]) + \text{EXTS}(X[i][1]) \times \text{EXTZ}(Y[j][1]) + \text{EXTS}(X[i][2]) \times \text{EXTZ}(Y[j][2]) + \text{EXTS}(X[i][3]) \times \text{EXTZ}(Y[j][3]) + \text{EXTS}(ACC[AT][i][j]) )
\]
VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturate Positive multiply, Positive accumulate XX3-form

xvi8ger4spp AT,XA,XB

Prefix Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturate Positive multiply, Positive accumulate MMIRR:XX3-form

pmxvi8ger4spp AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>9</th>
<th>//</th>
<th>PMSK</th>
<th>//</th>
<th>XMSK</th>
<th>YMSK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>8</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th>59</th>
<th>AT</th>
<th>//</th>
<th>A</th>
<th>B</th>
<th>99</th>
<th>ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX.Unavailable()

if "xvi8ger4pps" then do
  PMSK ← 0b1111
  XMSK ← 0b1111
  YMSK ← 0b1111
end
do i = 0 to 3
do j = 0 to 3
  if XMSK.bit[i] & YMSK.bit[j] then do
    prod0 ← (PMSK.bit[0]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].byte[0]) * EXTZ(VSR[32×BX+B].word[j].byte[0])
    prod1 ← (PMSK.bit[1]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].byte[1]) * EXTZ(VSR[32×BX+B].word[j].byte[1])
  psum ← prod0 + prod1 + prod2 + prod3
  ACC[AT][i].word[j] ← si32_CLAMP( psum + EXTS(ACC[AT][i].word[j]) )
  if sat_flag=1 then VSCR.SAT ← 1
else
  ACC[AT][i][j] ← 0x0000_0000
end
end

Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of 32-bit signed integer values.

For xvi8ger4spp, let PMSK=0b1111, XMSK=0b1111, and YMSK=0b1111.

For each integer value i from 0 to 3, and each integer value j from 0 to 3, do the following.
If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.
If bit 0 of PMSK is equal to 1, let prod0 be the product of the 8-bit signed integer value in byte 0 of word element i of VSR[XA] (X[i][0]) and the 8-bit unsigned integer value in byte 0 of word element j of VSR[XB] (Y[j][0]), sign-extended to 32 bits. Otherwise, let prod0 be the value 0.

If bit 1 of PMSK is equal to 1, let prod1 be the product of the 8-bit signed integer value in byte 1 of word element i of VSR[XA] (X[i][1]) and the 8-bit unsigned integer value in byte 1 of word element j of VSR[XB] (Y[j][1]), sign-extended to 32 bits. Otherwise, let prod1 be the value 0.
If bit 2 of PMSK is equal to 1, let prod2 be the product of the 8-bit signed integer value in byte 2 of word element i of VSR[XA](X[i][2]) and the 8-bit unsigned integer value in byte 2 of word element i of VSR[XB](Y[i][2]), sign-extended to 32 bits. Otherwise, let prod2 be the value 0.

If bit 3 of PMSK is equal to 1, let prod3 be the product of the 8-bit signed integer value in byte 3 of word element i of VSR[XA](X[i][3]) and the 8-bit unsigned integer value in byte 3 of word element i of VSR[XB](Y[i][3]), sign-extended to 32 bits. Otherwise, let prod3 be the value 0.

Let psum be the sum of prod0, prod1, prod2, and prod3. psum is added to the 32-bit signed integer value in word element j of ACC[AT][i] and the result is placed into word element j of ACC[AT][i] in 32-bit signed integer format.

If the result is less than $-2^{31}$, the result saturates to $-2^{31}$ and SAT is set to 1.
If the result is greater than $2^{31} - 1$, the result saturates to $2^{31} - 1$ and SAT is set to 1.

Otherwise, let word element j of ACC[AT][i] is set to 0x0000_0000.

**Special Registers Altered:**
SAT

**Register Operand Data Layout for [pm]xvi8ger4[pp]**

<table>
<thead>
<tr>
<th>VSR[XA]</th>
<th>VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>X[0]</td>
<td>Y[0]</td>
</tr>
<tr>
<td>X[0]</td>
<td>Y[0]</td>
</tr>
</tbody>
</table>

**Programming Note**
Let X be the 4x4 matrix of 8-bit signed integer values contained in VSR[XA] in row-major format.
Let Y be the 4x4 matrix of 8-bit unsigned integer values contained in VSR[XB] in row-major format.
Let ACC[AT] be the Accumulator containing a 4x4 matrix of 32-bit signed integer values.

[pm]xvi8ger4 (see <PAGE XREF to xvi8ger4>) performs the following form of accumulation of four outer products (rank 4 update).

ACC[i][j] = si32_CHOP( EXTS(X[i][0]) * EXTZ(Y[j][0]) + EXTS(X[i][1]) * EXTZ(Y[j][1]) + EXTS(X[i][2]) * EXTZ(Y[j][2]) + EXTS(X[i][3]) * EXTZ(Y[j][3]) )

Note that a saturating form of the above accumulation is not needed nor provided since the sum of four 16-bit signed integer products cannot overflow the 32-bit signed integer format.

[pm]xvi8ger4spp performs the following form of accumulation of four outer products (rank 4 update).

ACC[i][j] = si32_CLAMP( EXTS(X[i][0]) * EXTZ(Y[j][0]) + EXTS(X[i][1]) * EXTZ(Y[j][1]) + EXTS(X[i][2]) * EXTZ(Y[j][2]) + EXTS(X[i][3]) * EXTZ(Y[j][3]) + EXTS(ACC[i][j]) )
VSX Vector 16-bit Signed Integer GER (rank-2 update) XX3-form

```c
xvi16ger2 AT,XA,XB
```

VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate XX3-form

```c
xvi16ger2pp AT,XA,XB
```

Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) MMIRR:XX3-form

```c
pmxvi16ger2 AT,XA,XB,XMSK,YMSK,PMSK
```

Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form

```c
pmxvi16ger2pp AT,XA,XB,XMSK,YMSK,PMSK
```

```c
if MSR.VSX=0 then VSX_Unavailable();

if "xvi16ger2s" | "xvi16ger2spp" then do
    PMSK ← 0b11
    XMSK ← 0b1111
    YMSK ← 0b1111
end

i = 0 to 3
    j = 0 to 3
    if XMSK.bit[i] & YMSK.bit[j] then do
        prod0 ← (PMSK.bit[0]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].hword[0]) * EXTS(VSR[32×BX+B].word[j].hword[0])
        prod1 ← (PMSK.bit[1]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].hword[1]) * EXTS(VSR[32×BX+B].word[j].hword[1])
        psum ← prod0 + prod1
        if "[pm]xvi16ger2" then ACC[AT][i].word[j] ← CHOP32(psum)
        if "[pm]xvi16ger2pp" then ACC[AT][i].word[j] ← CHOP32(psum + EXTS(ACC[AT][i].word[j]))
    end
end
```

```c
ACC[AT][i][j] ← 0x0000_0000
end
```
Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid. Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of 32-bit signed integer values.

For xvi16ger2 or xvi16ger2pp, let PMSK=0b11, XMSK=0b1111, and YMSK=0b1111.

For each integer value i from 0 to 3 and each integer value j from 0 to 3, do the following.

If bit i of PMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.

If bit 0 of PMSK is equal to 1, let prod0 be the product of the 16-bit signed integer value in halfword 0 of word element i of VSR[XA] and the 16-bit signed integer value in halfword 0 of word element j of VSR[XB]. Otherwise, let prod0 be the value 0.

If bit 1 of PMSK is equal to 1, let prod1 be the product of the 16-bit signed integer value in halfword 1 of word element i of VSR[XA] and the 16-bit signed integer value in halfword 1 of word element j of VSR[XB]. Otherwise, let prod1 be the value 0.

Let psum be the sum of prod0 and prod1.

For [pm]xvi16ger2, psum is placed into word element j of ACC[AT][i] in 32-bit signed integer format.

For [pm]xvi16ger2pp, psum is added to the 32-bit signed integer value in word element j of ACC[AT][i], and the result is placed into word element j of ACC[AT][i] in 32-bit signed integer format.

Otherwise, let ACC[AT][i][j] is set to 0x0000_0000.

Special Registers Altered:

| None |

Register Operand Data Layout for [pm]xvi16ger2[pp]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X[0][0]</td>
<td>Y[0][0]</td>
<td>T[0][0]</td>
<td>T[1][0]</td>
<td>T[2][0]</td>
<td>T[3][0]</td>
</tr>
</tbody>
</table>

Programming Note

Let X be the 4×2 matrix of 16-bit signed integer values contained in VSR[XA] in row-major format. Let Y be the 4×2 matrix of 16-bit signed integer values contained in VSR[XB] in row-major format. Let ACC[AT] be the Accumulator containing a 4×4 matrix of 32-bit signed integer values.

[pm]xvi16ger2 performs the following form of accumulation of two outer products (rank 2 update).

\[
\text{ACC}[i][j] = \text{si32}_\text{CHOP}(\text{EXTS}(X[i][0]) \times \text{EXTS}(Y[j][0]) + \text{EXTS}(X[i][1]) \times \text{EXTS}(Y[j][1]) + \text{EXTS}(\text{ACC}[i][j]))
\]

[pm]xvi16ger2pp performs the following form of accumulation of two outer products (rank 2 update).

\[
\text{ACC}[i][j] = \text{si32}_\text{CHOP}(\text{EXTS}(X[i][0]) \times \text{EXTS}(Y[j][0]) + \text{EXTS}(X[i][1]) \times \text{EXTS}(Y[j][1]) + \text{EXTS}(\text{ACC}[i][j]))
\]
VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation XX3-form

```
xvi16ger2s  AT,XA,XB
```

VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate XX3-form

```
xvi16ger2spp  AT,XA,XB
```

Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation MMIRR:XX3-form

```
pxvi16ger2s  AT,XA,XMSK,YMSK,PMSK
```

Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form

```
pxvi16ger2spp  AT,XA,XMSK,YMSK,PMSK
```

if MSR.VSX=0 then VSX_Unavailable()

if "xvi16ger2s" | "xvi16ger2spp" then do
  PMSK ← 0b11
  XMSK ← 0b1111
  YMSK ← 0b1111
  end

sat_flag ← 0

do i = 0 to 3
  do j = 0 to 3
    if XMSK.bit[i] & YMSK.bit[j] then do
      prod0 ← (PMSK.bit[0]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].hword[0]) * EXTS(VSR[32×BX+B].word[j].hword[0])
      prod1 ← (PMSK.bit[1]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].hword[1]) * EXTS(VSR[32×BX+B].word[j].hword[1])
      psum ← prod0 + prod1
    end
    if "[pm]xvi16ger2s" then
      ACC[AT][i].word[j] ← si32_CLAMP( psum | pmxvi16ger2spp  AT,XA,XMSK,YMSK,PMSK
```

Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation MMIRR:XX3-form

```
pxvi16ger2s  AT,XA,XMSK,YMSK,PMSK
```

Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form

```
pxvi16ger2spp  AT,XA,XMSK,YMSK,PMSK
```

if sat_flag=1 then VSCR.SAT ← 1
Let $XA$ be the value of $32 \times AX + A$. If $XA$ is in the range $4 \times AT$ to $4 \times AT + 3$, the instruction form is invalid.
Let $XB$ be the value of $32 \times BX + B$. If $XB$ is in the range $4 \times AT$ to $4 \times AT + 3$, the instruction form is invalid.

Let the contents of $ACC[AT]$ be a $4 \times 4$ matrix of 32-bit signed integer values.

For $xvi16ger2s$ or $xvi16ger2spp$, let $PMSK=0b11$, $XMSK=0b1111$, and $YMSK=0b1111$.

For each integer value $i$ from 0 to 3 and each integer value $j$ from 0 to 3, do the following.

If bit $i$ of $XMSK$ is equal to 1 and bit $j$ of $YMSK$ is equal to 1, do the following.

If bit 0 of $PMSK$ is equal to 1, let $prod0$ be the product of the 16-bit signed integer value in halfword 0 of word element $i$ of $VSR[XA]$ and the 16-bit signed integer value in halfword 0 of word element $j$ of $VSR[XB]$.
Otherwise, let $prod0$ be the value 0.

If bit 1 of $PMSK$ is equal to 1, let $prod1$ be the product of the 16-bit signed integer value in halfword 1 of word element $i$ of $VSR[XA]$ and the 16-bit signed integer value in halfword 1 of word element $j$ of $VSR[XB]$.
Otherwise, let $prod1$ be the value 0.

Let $sum$ be the sum of $prod0$ and $prod1$.

For $[pm]xvi16ger2s$, let $result$ be $psum$.

For $[pm]xvi16ger2spp$, let $result$ be the sum of $psum$ to the 32-bit signed integer value in word element $j$ of $ACC[AT][i]$.

If $result$ is less than $-2^{31}$, $result$ saturates to $-2^{31}$ and SAT is set to 1.
If $result$ is greater than $2^{31} - 1$, $result$ saturates to $2^{31} - 1$ and SAT is set to 1.

$result$ is placed into word element $j$ of $ACC[AT][i]$ in 32-bit signed integer format.

Otherwise, let $ACC[AT][i][j]$ be set to $0x0000_0000$.

Special Registers Altered:

SAT

---

**Register Operand Data Layout for [pm]xvi16ger2s[pp]**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$X[0][0]$</td>
<td>$Y[0][0]$</td>
<td>$T[0][0]$</td>
<td>$T[1][0]$</td>
<td>$T[2][0]$</td>
<td>$T[3][0]$</td>
</tr>
</tbody>
</table>

0 16 32 48 64 80 96 112 127
Let $X$ be the $4 \times 2$ matrix of 16-bit signed integer values contained in $VSR[XA]$ in row-major format.
Let $Y$ be the $4 \times 2$ matrix of 16-bit signed integer values contained in $VSR[XB]$ in row-major format.
Let $ACC[AT]$ be the Accumulator containing a $4 \times 4$ matrix of 32-bit signed-integer values.

\[ \text{pm}xvi16ger2s \] performs the following form of accumulation of two outer products (rank 2 update).

\[
\text{ACC}[AT][i][j] = \text{si32_CLAMP}( \text{EXTS}(X[i][0]) \cdot \text{EXTS}(Y[j][0]) + \\
\text{EXTS}(X[i][1]) \cdot \text{EXTS}(Y[j][1]))
\]

\[ \text{pm}xvi16ger2spp \] performs the following form of accumulation of two outer products (rank 2 update).

\[
\text{ACC}[AT][i][j] = \text{si32_CLAMP}( \text{EXTS}(X[i][0]) \cdot \text{EXTS}(Y[j][0]) + \\
\text{EXTS}(X[i][1]) \cdot \text{EXTS}(Y[j][1]) + \\
\text{EXTS}(\text{ACC}[AT][i][1]))
\]
### VSX Vector Insert Exponent Double-Precision XX3-form

#### xviexpdp

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>248</th>
<th>VSR[XT].dword[0]</th>
<th>VSR[XT].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VSR Data Layout for xviexpdp**

- **src1**: VSR[XA].dword[0]
- **src2**: VSR[XB].dword[0]
- **tgt**: VSR[XT].dword[0]

### VSX Vector Insert Exponent Single-Precision XX3-form

#### xviexpsp

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VSR Data Layout for xviexpsp**

VSX Vector Multiply-Add Type-A
Double-Precision XX3-form

\[ \text{xvmaddadp} \ XT, XA, XB \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

VSX Vector Multiply-Add Type-M
Double-Precision XX3-form

\[ \text{xvmaddmdp} \ XT, XA, XB \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>105</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XA \) be the value \( 32 \times AX + A \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 1, do the following.

For \( \text{xvmaddadp} \), do the following.
- Let \( src1 \) be the double-precision floating-point operand in doubleword element \( i \) of \( VSR[XA] \).
- Let \( src2 \) be the double-precision floating-point operand in doubleword element \( i \) of \( VSR[XT] \).
- Let \( src3 \) be the double-precision floating-point operand in doubleword element \( i \) of \( VSR[XB] \).

For \( \text{xvmaddmdp} \), do the following.
- Let \( src1 \) be the double-precision floating-point operand in doubleword element \( i \) of \( VSR[XA] \).
- Let \( src2 \) be the double-precision floating-point operand in doubleword element \( i \) of \( VSR[XB] \).
- Let \( src3 \) be the double-precision floating-point operand in doubleword element \( i \) of \( VSR[XT] \).

\( src1 \) is multiplied\(^1\) by \( src3 \), producing a product having unbounded range and precision.

See part 1 of Table 126.

\( src2 \) is added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 126.

The intermediate result is rounded to double-precision using the rounding mode specified by \( RN \).


The result is placed into doubleword element \( i \) of \( VSR[XT] \) in double-precision format.

See Table 114, “Vector Floating-Point Final Result,” on page 823.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( VSR[XT] \).

Special Registers Altered
\[ FX \ OX \ UX \ VXSNAN \ VXISI \ VXIMZ \]
## VSR Data Layout for xvmaddadp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>

## VSR Data Layout for xvmaddmdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>
### Table 126. Actions for `xvmadd(a|m)`dp

<table>
<thead>
<tr>
<th><code>src3</code></th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>p ← dQNaN</td>
<td>p ← +Infinity</td>
<td>p ← -Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxisi_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>p ← +Infinity</td>
<td>p ← M(src1, src3)</td>
<td>p ← +Zero</td>
<td>p ← -Zero</td>
<td>p ← M(src1, src3)</td>
<td>p ← -Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Zero</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← dQNaN</td>
<td>p ← -Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Zero</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← -Infinity</td>
<td>p ← -Infinity</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← -Infinity</td>
<td>p ← -Infinity</td>
<td>p ← dQNaN</td>
<td>p ← dQNaN</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1 vxsnan_flag ← 1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

#### Part 2: Add

<table>
<thead>
<tr>
<th><code>src2</code></th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← A(src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← -Zero</td>
<td>v ← +Rez</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← +Rez</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← A(src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

#### Explanation:

| `src1` | The double-precision floating-point value in doubleword element `i` of `VSR[IA]` (where `i = 0, 1`).
| `src2` | For `xvmadaddp`, the double-precision floating-point value in doubleword element `i` of `VSR[XT]` (where `i = 0, 1`).
| `src3` | For `xvmadaddp`, the double-precision floating-point value in doubleword element `i` of `VSR[XB]` (where `i = 0, 1`).
| dQNaN | Default quiet NaN (`0x7FF8_0000_0000_0000`).
| NZF | Nonzero finite number.
| Rez | Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with nonzero finite number source operands.
| Q(x) | Return a QNaN with the payload of `x`.
| A(x, y) | Return the normalized sum of floating-point value `x` and floating-point value `y`, having unbounded range and precision.
| Note: If `x = -y`, `x` is considered to be an exact-zero-difference result (Rez).
| M(x, y) | Return the normalized product of floating-point value `x` and floating-point value `y`, having unbounded range and precision.
| p | The intermediate product having unbounded range and precision.
| v | The intermediate result having unbounded range and precision.
VSX Vector Multiply-Add Type-A
Single-Precision XX3-form

\[ \text{xvmaddasp} \ XT, XA, XB \]

\[ \begin{array}{cccccc}
0 & 6 & T & A & B & 65 \\
\end{array} \]

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XA \) be the value \( 32 \times AX + A \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 3, do the following.

For \( \text{xvmaddasp} \), do the following.

- Let \( src1 \) be the single-precision floating-point operand in word element \( i \) of \( VSR[AX] \).
- Let \( src2 \) be the single-precision floating-point operand in word element \( i \) of \( VSR[XT] \).
- Let \( src3 \) be the single-precision floating-point operand in word element \( i \) of \( VSR[XB] \).

src1 is multiplied\(^1\) by src3, producing a product having unbounded range and precision.

The sum is normalized\(^3\).

The intermediate result is rounded to single-precision using the rounding mode specified by \( RN \).

The result is placed into word element \( i \) of \( VSR[XT] \) in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( VSR[XT] \).

Special Registers Altered

\[ \begin{array}{cccc}
FX & OX & UX & VXSNAN & VXISI & VXIMZ \\
\end{array} \]

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for xvmaddasp

<table>
<thead>
<tr>
<th></th>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>tgt</th>
</tr>
</thead>
</table>

### VSR Data Layout for xvmaddmsp

<table>
<thead>
<tr>
<th></th>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>tgt</th>
</tr>
</thead>
</table>
### Table 127. Actions for xvmadd(a|m)sp

<table>
<thead>
<tr>
<th>src3</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>p vmnz_flag ← 1</td>
<td>p ← dQNaN</td>
<td>p vmnz_flag ← 1</td>
<td>p ← –Infinity</td>
<td>p ← src3</td>
</tr>
<tr>
<td>–NZF</td>
<td>p ← +Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Infinity</td>
<td>p ← src3</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← –Infinity</td>
<td>p ← dQNaN</td>
<td>p vmnz_flag ← 1</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p vmnz_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

### Explanation:

- **src1**: The single-precision floating-point value in word element \(i\) of VSR[ XB] (where \(i = \{0, 1, 2, 3\}\).

- **src2**: For xvmaddasp, the single-precision floating-point value in word element \(i\) of VSR[ XT] (where \(i = \{0, 1, 2, 3\}\). For xvmaddmsp, the single-precision floating-point value in word element \(i\) of VSR[ X] (where \(i = \{0, 1, 2, 3\}\).

- **src3**: For xvmaddasp, the single-precision floating-point value in word element \(i\) of VSR[ XB] (where \(i = \{0, 1, 2, 3\}\). For xvmaddmsp, the single-precision floating-point value in word element \(i\) of VSR[ X] (where \(i = \{0, 1, 2, 3\}\).

- **dQNaN**: Default quiet NaN (0x7FC0_0000).

- **NZF**: Nonzero finite number.

- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.

- **Q(x)**: Return a QNaN with the payload of \(x\).

- **A(x, y)**: Return the normalized sum of floating-point value \(x\) and floating-point value \(y\), having unbounded range and precision.

  **Note:** If \(x = -y\), \(v\) is considered to be an exact-zero-difference result (Rezd).

- **M(x, y)**: Return the normalized product of floating-point value \(x\) and floating-point value \(y\), having unbounded range and precision.

  - **p**: The intermediate product having unbounded range and precision.

  - **v**: The intermediate result having unbounded range and precision.
Let $XT$ be the value $32\times T + T$.
Let $XA$ be the value $32\times AX + A$.
Let $XB$ be the value $32\times BX + B$.

For each integer value $i$ from 0 to 1, do the following.
Let $src1$ be the double-precision floating-point operand in doubleword element $i$ of $VSR[AX]$.

Let $src2$ be the double-precision floating-point operand in doubleword element $i$ of $VSR[XB]$.

If $src1$ is greater than $src2$, $src1$ is placed into doubleword element $i$ of $VSR[XT]$ in double-precision format. Otherwise, $src2$ is placed into doubleword element $i$ of $VSR[XT]$ in double-precision format.

The maximum of $+0$ and $-0$ is $+0$. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN when $VE=0$ is that SNaN converted to a QNaN.

See Table 128.

If a trap-enabled exception occurs in any element of the vector, no results are written to $VSR[XT]$.

**Special Registers Altered**

<table>
<thead>
<tr>
<th>FX</th>
<th>VXSNAN</th>
</tr>
</thead>
</table>

**VSR Data Layout for xvmxdp**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
<tr>
<td>src1</td>
<td>–Infinity</td>
<td>–NZF</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>–Infinity</td>
<td>T(src1)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>–NZF</td>
<td>T(src1)</td>
<td>T(M(src1,src2))</td>
</tr>
<tr>
<td>–Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
</tbody>
</table>

Explanation:
- **src1** The double-precision floating-point value in doubleword element \( i \) of VSR[\( \text{XA} \)] (where \( i = \{0, 1\} \)).
- **src2** The double-precision floating-point value in doubleword element \( i \) of VSR[\( \text{XT} \)] (where \( i = \{0, 1\} \)).
- **NZF** Nonzero finite number.
- **Q(x)** Return a QNaN with the payload of \( x \).
- **M(x,y)** Return the greater of floating-point value \( x \) and floating-point value \( y \).
- **T(x)** The value \( x \) is placed in doubleword element \( i = \{0, 1\} \) of VSR[\( \text{XT} \)] in double-precision format.
- **fx(x)** If \( i \) is equal to 0, FX is set to 1, \( i \) is set to 1.

Table 128. Actions for xvmxdp
VSX Vector Maximum Single-Precision
XX3-form

vmaxsp XT,XA,XB

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.
Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
Let src2 be the single-precision floating-point operand in word element i of VSR[XB].
If src1 is greater than src2, src1 is placed into word element i of VSR[XT] in single-precision format. Otherwise, src2 is placed into word element i of VSR[XT] in single-precision format.

The maximum of +0 and –0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 129.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
FX VXSNAN

VSR Data Layout for vmaxsp

|------|----------------|----------------|----------------|----------------|

Let vresult.word[i] ← bfp32_MAXIMUM(src1, src2)
<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>–NZF</td>
<td>T(src1)</td>
<td>T(M(src1, src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>–Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(M(src1, src2))</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
</tbody>
</table>

**Explanation:**
- **src1** The single-precision floating-point value in word element i of VSR[XA] (where i = {0, 1, 2, 3}).
- **src2** The single-precision floating-point value in word element i of VSR[XT] (where i = {0, 1, 2, 3}).
- **NZF** Nonzero finite number.
- **Q(x)** Return a QNaN with the payload of x.
- **M(x, y)** Return the greater of floating-point value x and floating-point value y.
- **T(x)** The value x is placed in word element i (i = {0, 1, 2, 3}) of VSR[XT] in single-precision format.
- **fx(x)** If FPSCR.x is equal to 0, FPSCR.FX is set to 1. FPSCR.x is set to 1.

**Table 129. Actions for xvmaxsp**
VSX Vector Minimum Double-Precision
XX3-form

```
xvmindp XT,XA,XB

if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
    reset_xflags()
    src1 ← VSR[32×AX+A].dword[i]
    src2 ← VSR[32×BX+B].dword[i]
    vresult.dword[i] ← bfp64_MINIMUM(src1,src2)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end
if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.
Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src1 is less than src2, src1 is placed into doubleword element i of VSR[XT] in double-precision format. Otherwise, src2 is placed into doubleword element i of VSR[XT] in double-precision format.

The minimum of +0 and -0 is -0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 130.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

 FX VXSNAN

---

**VSR Data Layout for xvmindp**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>
### Table 130. Actions for xvmindp

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(src1)</td>
</tr>
<tr>
<td>–NZF</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>–Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
</tbody>
</table>

**Explanation:**
- **src1**: The double-precision floating-point value in doubleword element \( i \) of \( \text{VSR}[\text{XI}] \) (where \( i = \{0, 1\} \)).
- **src2**: The double-precision floating-point value in doubleword element \( i \) of \( \text{VSR}[\text{XT}] \) (where \( i = \{0, 1\} \)).
- **NZF**: Nonzero finite number.
- **Q(x)**: Return a QNaN with the payload of \( x \).
- **M(x,y)**: Return the lesser of floating-point value \( x \) and floating-point value \( y \).
- **T(x)**: The value \( x \) is placed in doubleword element \( i \) of \( \text{VSR}[\text{XT}] \) in double-precision format.
- **fx(x)**: If \( \text{FPSCR} > x \) is equal to 0, \( \text{FPSCR} > \text{FX} \) is set to 1. If \( \text{FPSCR} > x \) is not equal to 0, \( \text{FPSCR} > \text{FX} \) is set to 1.
VSX Vector Minimum Single-Precision
XX3-form

vxminsp XT,XA,XB

<table>
<thead>
<tr>
<th>T</th>
<th>A</th>
<th>B</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

doi = 0 to 3
reset_xflags()

src1 ← VSR[32×AX+A].word[i]
src2 ← VSR[32×BX+B].word[i]
vresult.word[i] ← bfp32_MINIMUM(src1,src2)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.
Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

If src1 is less than src2, src1 is placed into word element i of VSR[XT] in single-precision format.
Otherwise, src2 is placed into word element i of VSR[XT] in single-precision format.

The minimum of +0 and -0 is -0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 131.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
FX VXSNAN
<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>–NZF</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>–Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The single-precision floating-point value in word element i of VSR[XT] (where i = {0, 1, 2, 3}).
- **src2** The single-precision floating-point value in word element i of VSR[XA] (where i = {0, 1, 2, 3}).
- **NZF** Nonzero finite number.
- **Q** Return a QNaN with the payload of x.
- **M** Return the lesser of floating-point value x and floating-point value y.
- **T** The value x is placed in word element i (i = {0, 1, 2, 3}) of VSR[XT] in single-precision format.
- **Fx** If FPSCR.x is equal to 0, FPSCR.FX is set to 1. FPSCR.x is set to 1.
- **VXSNAN** Floating-point Invalid Operation Exception (SNaN). If FPSCR.VE=1, update of VSR[XT] is suppressed.

**Table 131. Actions for xvminsp**
VSX Vector Multiply-Subtract Type-A
Double-Precision XX3-form

\[
xvmsubadp \ XT,XA,XB
\]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>113</th>
<th>K</th>
<th>B</th>
<th>X</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>21</td>
<td>113</td>
<td>K</td>
<td>B</td>
<td>X</td>
<td>T</td>
</tr>
</tbody>
</table>

VSX Vector Multiply-Subtract Type-M
Double-Precision XX3-form

\[
xvmsubmdp \ XT,XA,XB
\]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>121</th>
<th>K</th>
<th>B</th>
<th>X</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>21</td>
<td>121</td>
<td>K</td>
<td>B</td>
<td>X</td>
<td>T</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
reset_xflags()
if "xvmsubadp" then do
src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
end
else do
src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
end
v ← bfp_MULTIPLY_ADD(src1,src3,bfp_NEGATE(src2))
rnd ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)
vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
| (FPSCR.VE & vxisi_flag)
| (FPSCR.VE & vximz_flag)
| (FPSCR.VE & ox_flag)
| (FPSCR.VE & ux_flag)
| (FPSCR.VE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← result

Let XT be the value 32×TX + T. Let XA be the value 32×AX + A. Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

For xvmsubadp, do the following.
- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 132.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 132.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.


The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 114, “Vector Floating-Point Final Result,” on page 823.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
FX OX UX XX VXSNAN VXISI VXIMZ

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for xvmsubadp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>

| 0 | 64 | 127 |

### VSR Data Layout for xvmsubmdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>

| 0 | 64 | 127 |
Table 132. Actions for `xvmsub(a|m)dp`

### Part 1: Multiply

<table>
<thead>
<tr>
<th><code>src3</code></th>
<th><code>-Infinity</code></th>
<th><code>-NZF</code></th>
<th><code>-Zero</code></th>
<th><code>+Zero</code></th>
<th><code>+NZF</code></th>
<th><code>+Infinity</code></th>
<th><code>QNaN</code></th>
<th><code>SNaN</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-Infinity</code></td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>v ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td><code>-NZF</code></td>
<td>p ← +Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td><code>-Zero</code></td>
<td>p ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td><code>+Zero</code></td>
<td>p ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td><code>+NZF</code></td>
<td>p ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td><code>+Infinity</code></td>
<td>p ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td><code>QNaN</code></td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td><code>SNaN</code></td>
<td>p ← Q(src1)</td>
<td>v ← xsnan_flag</td>
<td>v ← xsnan_flag</td>
<td>v ← xsnan_flag</td>
<td>v ← xsnan_flag</td>
<td>v ← xsnan_flag</td>
<td>v ← xsnan_flag</td>
<td>v ← xsnan_flag</td>
</tr>
</tbody>
</table>

### Part 2: Subtract

<table>
<thead>
<tr>
<th><code>src2</code></th>
<th><code>-Infinity</code></th>
<th><code>-NZF</code></th>
<th><code>-Zero</code></th>
<th><code>+Zero</code></th>
<th><code>+NZF</code></th>
<th><code>+Infinity</code></th>
<th><code>QNaN</code></th>
<th><code>SNaN</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-Infinity</code></td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td><code>-NZF</code></td>
<td>v ← +Infinity</td>
<td>v ← S(src2)</td>
<td>v ← +Infinity</td>
<td>v ← S(src2)</td>
<td>v ← +Infinity</td>
<td>v ← S(src2)</td>
<td>v ← +Infinity</td>
<td>v ← S(src2)</td>
</tr>
<tr>
<td><code>-Zero</code></td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td><code>+Zero</code></td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td><code>+NZF</code></td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td><code>+Infinity</code></td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td><code>QNaN &amp; src1 is a NaN</code></td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
<tr>
<td><code>QNaN &amp; src1 not a NaN</code></td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
</tbody>
</table>

### Explanation:

| `src1` | The double-precision floating-point value in doubleword element `i` of `VSR[AA]` (where `i` ∈ {0, 1}). |
| `src2` | For `xvmsubadp`, the double-precision floating-point value in doubleword element `i` of `VSR[XT]` (where `i` ∈ {0, 1}). |
| `src3` | For `xvmsubadp`, the double-precision floating-point value in doubleword element `i` of `VSR[XB]` (where `i` ∈ {0, 1}). |
| `dQNaN` | Default quiet NaN (`0x7FF8_0000_0000_0000`). |
| `NZF` | Nonzero finite number. |
| `Rezd` | Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands. |
| `Q(x)` | Return a QNaN with the payload of `x`. |
| `S(x,y)` | Return the normalized sum of floating-point value `x` and negated floating-point value `y`, having unbounded range and precision. Note: If `x = y`, `v` is considered to be an exact-zero-difference result (`Rezd`). |
| `M(x,y)` | Return the normalized product of floating-point value `x` and floating-point value `y`, having unbounded range and precision. |
| `p` | The intermediate product having unbounded range and precision. |
| `v` | The intermediate result having unbounded range and precision. |
**VSX Vector Multiply-Subtract Type-A**

**Single-Precision XX3-form**

\[
\text{xvmsubasp} \rightarrow XT, XA, XB
\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>81</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

**VSX Vector Multiply-Subtract Type-M**

**Single-Precision XX3-form**

\[
\text{xvmsubmsp} \rightarrow XT, XA, XB
\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable();

\[ex\_flag \leftarrow 0b0\]

do i = 0 to 3

reset_xflags();

if "xvmsubasp" then do

\[\text{src1} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP32}(\text{VSR}[32\times AX + A].\text{word}[i])\]
\[\text{src2} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP32}(\text{VSR}[32\times TX + T].\text{word}[i])\]
\[\text{src3} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP32}(\text{VSR}[32\times BX + B].\text{word}[i])\]
end

else do

\[\text{src1} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP32}(\text{VSR}[32\times AX + A].\text{word}[i])\]
\[\text{src2} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP32}(\text{VSR}[32\times BX + B].\text{word}[i])\]
\[\text{src3} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP32}(\text{VSR}[32\times TX + T].\text{word}[i])\]
end

\[v \leftarrow \text{bfp\_MULTIPLY\_ADD}(\text{src1}, \text{src3}, \text{bfp\_NEGATE}(\text{src2}))\]
\[\text{rnd} \leftarrow \text{bfp\_ROUND\_TO\_BFP32}(\text{FPSCR}.RN, v)\]
\[\text{vresult}.\text{word}[i] \leftarrow \text{bfp32\_CONVERT\_FROM\_BFP}(\text{rnd})\]

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxisn_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

\[\text{ex\_flag} \leftarrow \text{ex\_flag} | \text{FPSCR.VE \& vxsnan\_flag} | \text{FPSCR.VE \& vxisn\_flag} | \text{FPSCR.VE \& ox\_flag} | \text{FPSCR.VE \& ux\_flag} | \text{FPSCR.VE \& xx\_flag}\]

end

if ex\_flag=0 then VSR[32\times TX + T] \leftarrow \text{result}

**Special Registers Altered**

FX \text{OX \& UX \& VXSNAN \& VXISI \& VXI MW}

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent incremented by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for xvmsubasp

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tgt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|------|----------------|----------------|----------------|-----------------|

### VSR Data Layout for xvmsubmsp

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tgt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|------|----------------|----------------|----------------|-----------------|
### Table 133. Actions for xvmsub(a|m)sp

#### Part 1: Multiply

<table>
<thead>
<tr>
<th>src3</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>p ← –Infinity</td>
<td>p ← –Infinity</td>
<td>p ← dQNaN v</td>
<td>–Infinity</td>
<td>p ← –Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vximz_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td>p ← –Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Zero</td>
<td>p ← –Zero</td>
<td>p ← –Zero</td>
<td>p ← dQNaN v</td>
<td>p ← src3</td>
<td>p ← Q(src3) vximz_flag ← 1</td>
</tr>
<tr>
<td>–Zero</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← +Zero</td>
<td>p ← –Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← src3</td>
<td>p ← Q(src3) vximz_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← –Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← –Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← src3</td>
<td>p ← Q(src3) vximz_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← –Infinity</td>
<td>p ← –Infinity</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vximz_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

#### Part 2: Subtract

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>v ← dQNaN vximl_flag ← 1</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vximl_flag ← 1</td>
</tr>
<tr>
<td>–NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(p,src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← S(p,src2)</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vximl_flag ← 1</td>
</tr>
<tr>
<td>–Zero</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← S(p,src2)</td>
<td>v ← S(p,src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← S(p,src2)</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vximl_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← –Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
</tbody>
</table>

#### Explanation:

- **src1**: The single-precision floating-point value in word element \(i\) of \(VSR[XA]\) (where \(i = \{0, 1, 2, 3\}\).
- **src2**: For \(xvmsubsp\), the single-precision floating-point value in word element \(i\) of \(VSR[XT]\) (where \(i = \{0, 1, 2, 3\}\)).
- **src3**: For \(xvmsubsp\), the single-precision floating-point value in word element \(i\) of \(VSR[XA]\) (where \(i = \{0, 1, 2, 3\}\)).
- **dQNaN**: Default quiet NaN (\(\times[7F]0000\)).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)**: Return a QNaN with the payload of \(x\).
- **S(x, y)**: Return the normalized sum of floating-point value \(x\) and negated floating-point value \(y\), having unbounded range and precision. Note: If \(x = y\), \(x\) is considered to be an exact-zero-difference result (\(\times[00]\)).
- **M(x, y)**: Return the normalized product of floating-point value \(x\) and floating-point value \(y\), having unbounded range and precision.
- **v**: The intermediate product having unbounded range and precision.
- **p**: The intermediate result having unbounded range and precision.

---

**Part 3: VSR instruction**

- **VSR[XA]**
- **VSR[XT]**

---

**Power ISA™ I**
VSX Vector Multiply Double-Precision

XX3-form

```
xvmuldp XT,XA,XB
```

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>112</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>112</td>
<td>60</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

\[
\text{ex_flag} \leftarrow 0b0
\]

\[
do i = 0 to 1
\]

\[
\text{reset_xflags()}
\]

\[
\text{src1} \leftarrow \text{bfp\_CONVERT\_FROM\_BF64(VSR[32\times XA+A].dword[i])}
\]

\[
\text{src3} \leftarrow \text{bfp\_CONVERT\_FROM\_BF64(VSR[32\times XB+B].dword[i])}
\]

\[
\text{v} \leftarrow \text{bfp\_MULTIPLY(src1,src3)}
\]

\[
\text{rnd} \leftarrow \text{bfp\_ROUND\_TO\_BF64(0b0,FPSCR.RN,v)}
\]

\[
\text{vresult.dword[i]} \leftarrow \text{bfp64\_CONVERT\_FROM\_BF64(rnd)}
\]

\[
\text{if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)}
\]

\[
\text{if vximz_flag=1 then SetFX(FPSCR.VXIMZ)}
\]

\[
\text{if ox_flag=1 then SetFX(FPSCR.OX)}
\]

\[
\text{if ux_flag=1 then SetFX(FPSCR.UX)}
\]

\[
\text{if xx_flag=1 then SetFX(FPSCR.XX)}
\]

\[
\text{ex_flag} \leftarrow \text{ex_flag | (FPSCR.VE & vxsnan_flag)}
\]

\[
\text{ex_flag} \leftarrow \text{ex_flag | (FPSCR.VE & vximz_flag)}
\]

\[
\text{ex_flag} \leftarrow \text{ex_flag | (FPSCR.OE & ox_flag)}
\]

\[
\text{ex_flag} \leftarrow \text{ex_flag | (FPSCR.UE & ux_flag)}
\]

\[
\text{ex_flag} \leftarrow \text{ex_flag | (FPSCR.XE & xx_flag)}
\]

end

\[
\text{if ex_flag=0 then VSR[32\times TX+T] \leftarrow vresult}
\]

Let XT be the value \(32\times TX + T\).
Let XA be the value \(32\times XA + A\).
Let XB be the value \(32\times XB + B\).

For each integer value \(i\) from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element \(i\) of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element \(i\) of VSR[XB].

src1 is multiplied\(^1\) by src2, producing a product having unbounded range and precision.

The product is normalized\(^2\).

See Table 134.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.


The result is placed into doubleword element \(i\) of VSR[XT] in double-precision format.

See Table 114, “Vector Floating-Point Final Result,” on page 823.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXIMZ

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← Infinity</td>
<td>v ← Infinity</td>
<td>v ← NaN</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← Infinity</td>
<td>v ← M(src1, src2)</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← NaN</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← NaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← NaN</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← NaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← -Infinity</td>
<td>v ← M(src1, src2)</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

**Table 134. Actions for xvmuldp**

**Explanation:**
- **src1** The double-precision floating-point value in doubleword element i of VSR[XA] (where i = 0, 1).
- **src2** The double-precision floating-point value in doubleword element i of VSR[XB] (where i = 0, 1).
- **NaN** Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF** Nonzero finite number.
- **M(x,y)** Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **Q(x)** Return a QNaN with the payload of x.
- **v** The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Vector Multiply Single-Precision
XX3-form

\[ xvmulsp \quad XT,XA,XB \]

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XA \) be the value \( 32 \times AX + A \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 3, do the following.
Let \( src1 \) be the single-precision floating-point operand in word element \( i \) of \( VSR[XA] \).
Let \( src2 \) be the single-precision floating-point operand in word element \( i \) of \( VSR[XB] \).

\( src1 \) is multiplied\(^1\) by \( src2 \), producing a product having unbounded range and precision.

The product is normalized\(^2\).

See Table 135.

The intermediate result is rounded to single-precision using the rounding mode specified by \( RN \).


The result is placed into word element \( i \) of \( VSR[XT] \) in single-precision format.

See Table 114, “Vector Floating-Point Final Result,” on page 823.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( VSR[XT] \).

**Special Registers Altered**
\[ FX \quad OX \quad UX \quad XX \quad VXSNAN \quad VXIMZ \]

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 135. Actions for xvmulsp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← M(src1, src2)</td>
<td>v ← +Zero</td>
<td>v ← t</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← M(src1, src2)</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The single-precision floating-point value in word element \(i\) of VSR[XA] (where \(i = \{0, 1, 2, 3\}\)).
- **src2** The single-precision floating-point value in word element \(i\) of VSR[XB] (where \(i = \{0, 1, 2, 3\}\)).
- **dQNaN** Default quiet NaN (0x7FC0_0000).
- **NZF** Nonzero finite number.
- **M(x, y)** Return the normalized product of floating-point value \(x\) and floating-point value \(y\), having unbounded range and precision.
- **Q(x)** Return a QNaN with the payload of \(x\).
- **v** The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Vector Negative Absolute
Double-Precision XX2-form

xvnabsdp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>III</th>
<th>B</th>
<th>489</th>
<th>B(7:0)</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

for i = 0 to 1
    src ← VSR[32×BX+B].dword[i]
    VSR[32×TX+T].dword[i] ← bfp64_NEGATIVE_ABSOLUTE(src)
end

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.
    The contents of doubleword element i of VSR[XB],
    with bit 0 set to 1, is placed into doubleword
    element i of VSR[XT].

Special Registers Altered
None

VSR Data Layout for xvnabsdp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>VSR[XB].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
<tr>
<td></td>
<td>0 64</td>
<td>127</td>
</tr>
</tbody>
</table>

VSR Data Layout for xvnabssp

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 32 64 96</td>
<td>127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VSX Vector Negate Double-Precision XX2-form

xvnegdp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>505</th>
<th>B</th>
<th>T</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();

do i = 0 to 1
  src ← VSR[32×BX+B].dword[i]
  VSR[32×TX+T].dword[i] ← bfp64_NEGATE(src)
end

Let XT be the value \(32\times TX + T\).
Let XB be the value \(32\times BX + B\).

For each integer value \(i\) from 0 to 1, do the following.
The contents of doubleword element \(i\) of VSR[XB], with bit 0 complemented, is placed into doubleword element \(i\) of VSR[XT].

Special Registers Altered
None

VSR Data Layout for xvnegdp

src

<table>
<thead>
<tr>
<th>VSR[XB].dword[0]</th>
<th>VSR[XB].dword[1]</th>
</tr>
</thead>
</table>

tgt

<table>
<thead>
<tr>
<th>VSR[XT].dword[0]</th>
<th>VSR[XT].dword[1]</th>
</tr>
</thead>
</table>

VSX Vector Negate Single-Precision XX2-form

xvnegsp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>441</th>
<th>B</th>
<th>T</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();

do i = 0 to 3
  src ← VSR[32×BX+B].word[i]
  VSR[32×TX+T].word[i] ← bfp32_NEGATE(src)
end

Let XT be the value \(32\times TX + T\).
Let XB be the value \(32\times BX + B\).

For each integer value \(i\) from 0 to 3, do the following.
The contents of word element \(i\) of VSR[XB], with bit 0 complemented, is placed into word element \(i\) of VSR[XT].

Special Registers Altered
None

VSR Data Layout for xvnegsp

src

|------------------|------------------|------------------|------------------|

tgt

|------------------|------------------|------------------|------------------|
VSX Vector Negative Multiply-Add Type-A
Double-Precision XX3-form

\texttt{xvmnaddap} XT,XA,XB

\begin{tabular}{|c|c|c|c|c|}
\hline
60 & T & A & B & 225 \\
\hline
0 & 6 & 11 & 16 & 21 \\
\hline
\end{tabular}

VSX Vector Negative Multiply-Add Type-M
Double-Precision XX3-form

\texttt{xvmnaddmdp} XT,XA,XB

\begin{tabular}{|c|c|c|c|c|}
\hline
60 & T & A & B & 233 \\
\hline
0 & 6 & 11 & 16 & 21 \\
\hline
\end{tabular}

if MSR.VSX=0 then VSXUnavailable()
x Blazers ← 0

do i = 0 to 1
reset_xflags()
if "xvmnaddap" then do
src1 ← bfp\_CONVERT\_FROM\_BFP64(VSR[32×AX+A].dword[i])
src2 ← bfp\_CONVERT\_FROM\_BFP64(VSR[32×TX+T].dword[i])
src3 ← bfp\_CONVERT\_FROM\_BFP64(VSR[32×BX+B].dword[i])
end
else do
src1 ← bfp\_CONVERT\_FROM\_BFP64(VSR[32×AX+A].dword[i])
src2 ← bfp\_CONVERT\_FROM\_BFP64(VSR[32×BX+B].dword[i])
src3 ← bfp\_CONVERT\_FROM\_BFP64(VSR[32×TX+T].dword[i])
end

v ← bfp\_MULTIPLY\_ADD(src1,src3,src2)
rnd ← bfp\_NEGATE(bfp\_ROUND\_TO\_BFP64(FPSCR.RN,v))
vresult.dword[i] ← bfp64\_CONVERT\_FROM\_BFP(rnd)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

\texttt{ex\_flag} ← \texttt{ex\_flag} | (FPSCR.VE & vxsnan_flag)
| (FPSCR.VE & vximz_flag)
| (FPSCR.VE & ox_flag)
| (FPSCR.VE & ux_flag)
| (FPSCR.VE & xx_flag)

if ex_flag=0 then VSR[32×TX+T] ← vresult

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

For \texttt{xvmnaddap}, do the following.
– Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
– Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
– Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

For \texttt{xvmnaddmdp}, do the following.
– Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
– Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].
– Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XT].

src1 is multiplied\(^1\) by src3, producing a product having unbounded range and precision.

See part 1 of Table 136.

src2 is added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 136.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.


The result is negated and placed into doubleword element i of VSR[XT] in double-precision format.

See Table 137, “Vector Floating-Point Final Result with Negation,” on page 926.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for xvnmaddadp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>

| 0 | 64 | 127 |

### VSR Data Layout for xvnmaddmdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>

| 0 | 64 | 127 |
Table 136. Actions for \texttt{xvmadd(a|m)dp}

<table>
<thead>
<tr>
<th>src1</th>
<th>Multiply</th>
<th>Add</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-Infinity</td>
<td>-NZF</td>
</tr>
<tr>
<td>-Infinity</td>
<td>p + Infinity</td>
<td>p + Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>p + Infinity</td>
<td>p + M(s src1, src3)</td>
</tr>
<tr>
<td>-Zero</td>
<td>p + QNaN</td>
<td>v + Zero</td>
</tr>
<tr>
<td>+Zero</td>
<td>p + QNaN</td>
<td>v + Zero</td>
</tr>
<tr>
<td>+NZF</td>
<td>p + QNaN</td>
<td>v + Zero</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p + QNaN</td>
<td>v + Zero</td>
</tr>
<tr>
<td>QNaN</td>
<td>p + src1</td>
<td>p + src1</td>
</tr>
<tr>
<td>NaN</td>
<td>p + Q(src1) vxsnan_flag ← 1</td>
<td>p + Q(src1) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1:** The double-precision floating-point value in doubleword element \( i \) of VSR[xt] (where \( i = [0,1] \)).
- **src2:** For \texttt{xvmadddp}, the double-precision floating-point value in doubleword element \( i \) of VSR[xt] (where \( i = [0,1] \)).
- **src3:** For \texttt{xvmadddp}, the double-precision floating-point value in doubleword element \( i \) of VSR[xt] (where \( i = [0,1] \)).
- **dQNaN:** Default quiet NaN (\( 0x7FF0_0000_0000_0000 \)).
- **NZF:** Nonzero finite number.
- **Rezd:** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x):** Return a QNaN with the payload of \( x \).
- **M(x, y):** Return the normalized sum of floating-point value \( x \) and floating-point value \( y \), having unbounded range and precision.
- **p:** The intermediate product having unbounded range and precision.
- **v:** The intermediate result having unbounded range and precision.
### Table 137. Vector Floating-Point Final Result with Negation

| Case | VE | OE | UE | ZE | XE | vxsnan_flag | vximz_flag | vxisi_flag | Is r inaccurate? (r ≠ v) | Is r incremented? (|r| > |v|) | Is q incremented? (|q| > |v|) | Returned Results and Status Setting |
|------|----|----|----|----|----|--------------|------------|------------|--------------------------|-----------------|-----------------|----------------------------------|
| Special |   |    |    |    |    |              |            |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 1  | 1  | 1            | 0          |            |                          |                 |                 | T(r), fx(VXISI)               |
|       | 0  | 0  | 0  | 1  | 1  | 0            | 1          |            |                          |                 |                 | T(r), fx(VXIMZ)               |
|       | 0  | 1  | 1  | 0  | 1  | 1            | 1          |            |                          |                 |                 | T(r), fx(VXSNAN)               |
|       | 1  | 1  | 0  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(r), fx(VXSNAN), fx(VXIMZ)   |
|       | 1  | 1  | 0  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(r), fx(VXSNAN), fx(VXISI), error() |
|       | 1  | 1  | 0  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(r), fx(VXISI), error()     |
|       | 1  | 1  | 0  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(r), fx(VXIMZ), error()     |
|       | 1  | 1  | 0  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(r), fx(VXSNAN), error()    |
|       | 1  | 1  | 0  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(r), fx(VXSNAN), fx(VXIMZ), error() |
| Normal |   |    |    |    |    |              |            |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |
|       | 1  | 1  | 1  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(N(r))                     |
|       | 1  | 1  | 1  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |
|       | 1  | 1  | 1  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(N(r))                     |
|       | 1  | 1  | 1  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(N(r))                     |
| Overflow |   |    |    |    |    |              |            |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |
|       | 1  | 1  | 1  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(N(r))                     |
|       | 1  | 1  | 1  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(N(r))                     |
|       | 1  | 1  | 1  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(N(r))                     |
|       | 1  | 1  | 1  | 1  | 1  | 1            | 1          |            |                          |                 |                 | T(N(r))                     |
| Tiny |   |    |    |    |    |              |            |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |
|       | 0  | 0  | 0  | 0  | 0  | 0            | 0          |            |                          |                 |                 | T(N(r))                     |

### Explanation:

- The results do not depend on this condition.
- error() The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
- fx(x) FPSCR.x is set to 1 if FPSCR.x=0.
- q The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target precision, unbounded exponent range.
- r The value defined in Table 62, “Scalar Floating-Point Intermediate Result Handling,” on page 661, significand rounded to the target precision, bounded exponent range.
- v The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.
- N(x) The value x is negated by complementing the sign bit of x.
- T(x) The value x is placed in element i of VSR[XT] in the target precision format (where i=x[0,1] for results with 64-bit elements, and i=x[0,1,3,4] for results with 32-bit elements).
**VSX Vector Negative Multiply-Add Type-A**
**Single-Precision XX3-form**

```
xvmaddasp  XT,XA,XB
```

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>193</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

**VSX Vector Negative Multiply-Add Type-M**
**Single-Precision XX3-form**

```
xvmaddmsp  XT,XA,XB
```

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>201</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
ex_flag ← 0b0
do i = 0 to 3
reset_xflags()
if "xvnmaddasp" then do
src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×TX+T].word[i])
src3 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
end
else do
src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
src3 ← bfp_CONVERT_FROM_BFP32(VSR[32×TX+T].word[i])
end
v ← bfp_MULTIPLY_ADD(src1,src3,src2)
rnd ← bfp_NEGATE(bfp_ROUND_TO_BFP32(FPSCR.RN,v))
vresult.word[i] ← bfp32_CONVERT_FROM_BFP32(rnd)
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)
ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
| (FPSCR.VE & vximz_flag)
| (FPSCR.VE & vxisi_flag)
| (FPSCR.OE & ox_flag)
| (FPSCR.UE & ux_flag)
| (FPSCR.XE & xx_flag)
end
if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

For `xvmaddasp`, do the following.
  - Let src1 be the single-precision floating-point operand in word element i of VSR[AX].
  - Let src2 be the single-precision floating-point operand in word element i of VSR[XT].
  - Let src3 be the single-precision floating-point operand in word element i of VSR[XB].

For `xvmaddmsp`, do the following.
  - Let src1 be the single-precision floating-point operand in word element i of VSR[AX].
  - Let src2 be the single-precision floating-point operand in word element i of VSR[XB].
  - Let src3 be the single-precision floating-point operand in word element i of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 138.

src2 is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 138.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.


The result is negated and placed into word element i of VSR[XT] in single-precision format.

See Table 137, “Vector Floating-Point Final Result with Negation,” on page 926.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

### Special Registers Altered

- FX
- OX
- UX
- VXSNAN
- VXISI
- VXIMZ

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for xvnmaddasp

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>tgt</th>
</tr>
</thead>
</table>

### VSR Data Layout for xvnmaddmsp

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>tgt</th>
</tr>
</thead>
</table>
### Table 138. Actions for xvmmadd(a|m)sp

<table>
<thead>
<tr>
<th>src</th>
<th>src3</th>
<th>(-\infty)</th>
<th>(-NZF)</th>
<th>(-Zero)</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>p + (-\infty)</td>
<td>p + (-\infty)</td>
<td>p + dQNan</td>
<td>p + dQNan</td>
<td>p + dQNan</td>
<td>p + (-\infty)</td>
<td>p + (-\infty)</td>
<td>p (-\infty)</td>
<td>p (-\infty)</td>
</tr>
<tr>
<td>(-NZF)</td>
<td>p + (-\infty)</td>
<td>p + M(src1,src3)</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + M(src1,src3)</td>
<td>p + (-\infty)</td>
<td>p + (-\infty)</td>
<td>p + (-\infty)</td>
<td>p + (-\infty)</td>
</tr>
<tr>
<td>+NZF</td>
<td>p + +Zero</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p + dQNan</td>
<td>p + dQNan</td>
<td>p + dQNan</td>
<td>p + dQNan</td>
<td>p + dQNan</td>
<td>p + dQNan</td>
<td>p + dQNan</td>
<td>p + dQNan</td>
<td>p + dQNan</td>
</tr>
<tr>
<td>QNaN</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
<td>p + src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p + Q(src1)</td>
<td>p + Q(src1)</td>
<td>p + Q(src1)</td>
<td>p + Q(src1)</td>
<td>p + Q(src1)</td>
<td>p + Q(src1)</td>
<td>p + Q(src1)</td>
<td>p + Q(src1)</td>
<td>p + Q(src1)</td>
</tr>
</tbody>
</table>

### Explanation:

- **src1**: The single-precision floating-point value in word element i of VSR[XA] (where i=0,1,2,3).
- **src2**: For **xvmmaddsp**, the single-precision floating-point value in word element i of VSR[XT] (where i=0,1,2,3).
- **src3**: For **xvmmaddsp**, the single-precision floating-point value in word element i of VSR[XB] (where i=0,1,2,3).
- **dQNan**: Default quiet NaN (0x3F00_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number operands.
- **Q(x)**: Return a QNaN with the payload of x.
- **A(x,y)**: Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
- **M(x,y)**: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p**: The intermediate product having unbounded range and precision.
- **v**: The intermediate result having unbounded range and precision.
### VSX Vector Negative Multiply-Subtract Type-A Double-Precision XX3-form

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>241</th>
<th>VSR[XT]</th>
</tr>
</thead>
</table>

#### VSX Vector Negative Multiply-Subtract Type-M Double-Precision XX3-form

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>249</th>
<th>VSR[XT]</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0
do i = 0 to 1
  reset_xflags()
  if "xvnmsubadp" then do
    src1 ← bfp CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
    src2 ← bfp CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
    src3 ← bfp CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
  end
  else do
    src1 ← bfp CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
    src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
    src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
  end
  v ← bfp MULTIPLY_ADD(src1,src3,bfp NEGATE(src2))
  rnd ← bfp NEGATE(bfp ROUND_TO_BFP64(FPSCR.RN,v))
  vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)
  if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
  if vxisi_flag=1 then SetFX(FPSCR.VXISI)
  if ox_flag=1 then SetFX(FPSCR.OX)
  if ux_flag=1 then SetFX(FPSCR.UX)
  if xx_flag=1 then SetFX(FPSCR.XX)
  ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
  | (FPSCR.VE & vxisi_flag)
  | (FPSCR.OE & ox_flag)
  | (FPSCR.UE & ux_flag)
  | (FPSCR.XE & xx_flag)
  end
  if ex_flag=0 then VSR[32×TX+T] ← vresult

#### Special Registers Altered

- FX
- UX
- VXSNAN
- VXISI
- VXIMZ

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

Let XT be the value $32 \times TX + T$.
Let XA be the value $32 \times AX + A$.
Let XB be the value $32 \times BX + B$.

For each integer value i from 0 to 1, do the following.

For `xvnmsubadp`, do the following.
- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

For `xvnmsubmdp`, do the following.
- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 139.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 139.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.


The result is negated and placed into doubleword element i of VSR[XT] in double-precision format.

See Table 137, “Vector Floating-Point Final Result with Negation,” on page 926.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].
### VSR Data Layout for xvnmsubadp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>

### VSR Data Layout for xvnmsubmdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>
Table 139. Actions for xvnmsub(a|m)dp

<table>
<thead>
<tr>
<th>src3</th>
<th>(\infty)</th>
<th>(-\infty)</th>
<th>(-\text{NZF})</th>
<th>0</th>
<th>+\text{NZF}</th>
<th>+\text{Infinity}</th>
<th>\text{QNaN}</th>
<th>S\text{NaN}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\text{Infinity})</td>
<td>p ← +\text{Infinity}</td>
<td>p ← +\text{Infinity}</td>
<td>p ← d\text{QNaN} vximz_flag ← 1</td>
<td>p ← d\text{QNaN} vximz_flag ← 1</td>
<td>p ← +\text{Infinity}</td>
<td>p ← +\text{Infinity}</td>
<td>p ← +\text{Infinity}</td>
<td>p ← +\text{Infinity}</td>
</tr>
<tr>
<td>(-\text{NZF})</td>
<td>p ← +\text{Infinity}</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← M(src1,src3)</td>
<td>p ← +\text{Infinity}</td>
<td>p ← +\text{Infinity}</td>
<td>p ← +\text{Infinity}</td>
</tr>
<tr>
<td>\text{QNaN} &amp; src1 is a NaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>\text{QNaN} &amp; src1 is not a NaN</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

Explanations:

src1  The double-precision floating-point value in doubleword element i of VSR[XA] (where i={0,1}).
src2  For xvnmsubadp, the double-precision floating-point value in doubleword element i of VSR[XT] (where i={0,1}).
src3  For xvnmsubadp, the double-precision floating-point value in doubleword element i of VSR[XB] (where i={0,1}).
For xvnmsubdmp, the double-precision floating-point value in doubleword element i of VSR[XT] (where i={0,1}).
d\text{QNaN}  Default quiet NaN (0x7FFE_0000_0000_0000).
NZF  Nonzero finite number.
Rezd  Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)  Return a QNaN with the payload of x.
S(x,y)  Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)  Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p  The intermediate product having unbounded range and precision.
v  The intermediate result having unbounded range and precision.
VSX Vector Negative Multiply-Subtract Type-A
Single-Precision XX3-form

Let XT be the value $32 \times TX + T$.
Let XA be the value $32 \times AX + A$.
Let XB be the value $32 \times BX + B$.

For each integer value $i$ from 0 to 3, do the following.

For \textit{xvnmsubasp}, do the following.
- Let src1 be the single-precision floating-point operand in word element $i$ of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element $i$ of VSR[XT].
- Let src3 be the single-precision floating-point operand in word element $i$ of VSR[XB].

src1 is multiplied\(^1\) by src3, producing a product having unbounded range and precision.

See part 1 of Table 140.

ds rc2 is negated and added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 140.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.


The result is negated and placed into word element $i$ of VSR[XT] in single-precision format.

See Table 137, "Vector Floating-Point Final Result with Negation," on page 926.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

\begin{itemize}
  \item \textbf{Special Registers Altered:} FX OX UX XX VXSNA VXISI VXIMZ
\end{itemize}

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for `xvnmsubasp`

|------|----------------|----------------|----------------|----------------|

### VSR Data Layout for `xvnmsubmsp`

|------|----------------|----------------|----------------|----------------|
### Table 140. Actions for \( \text{xnmsub}(a|m)\)sp

<table>
<thead>
<tr>
<th>src3</th>
<th>(-\infty)</th>
<th>(-\text{NZF})</th>
<th>(-\text{Zero})</th>
<th>(+\text{Zero})</th>
<th>(+\text{NZF})</th>
<th>(+\infty)</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>(-\infty)</td>
<td>(-\infty)</td>
<td>(-\text{NZF})</td>
<td>(-\text{Zero})</td>
<td>(-\text{Zero})</td>
<td>(-\text{Zero})</td>
<td>(-\text{Zero})</td>
<td>(-\text{Zero})</td>
</tr>
<tr>
<td>(-\text{NZF})</td>
<td>(-\text{NZF})</td>
<td>(-\text{NZF})</td>
<td>(-\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
</tr>
<tr>
<td>(-\text{Zero})</td>
<td>(-\text{Zero})</td>
<td>(-\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
</tr>
<tr>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
<td>(+\text{Zero})</td>
</tr>
<tr>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
<td>(+\text{NZF})</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>QNaN</td>
<td>QNaN</td>
<td>QNaN</td>
<td>QNaN</td>
<td>QNaN</td>
<td>QNaN</td>
<td>QNaN</td>
</tr>
<tr>
<td>SNaN</td>
<td>SNaN</td>
<td>SNaN</td>
<td>SNaN</td>
<td>SNaN</td>
<td>SNaN</td>
<td>SNaN</td>
<td>SNaN</td>
<td>SNaN</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The single-precision floating-point value in word element \( i \) of VSR[XA] (where \( i \in \{0,1,2,3\} \)).
- **src2** The single-precision floating-point value in word element \( i \) of VSR[XT] (where \( i \in \{0,1,2,3\} \)).
- **src3** The single-precision floating-point value in word element \( i \) of VSR[XB] (where \( i \in \{0,1,2,3\} \)).
- **dQNaN** Default quiet NaN \( (0x\text{FC}0\_0000) \).
- **NZF** Nonzero finite number.
- **Rezd** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)** Return a QNaN with the payload of \( x \).
- **S(x,y)** Return the normalized sum of floating-point value \( x \) and negated floating-point value \( y \), having unbounded range and precision.
  - **Note:** If \( x = -y \), \( v \) is considered to be an exact-zero-difference result (Rezd).
- **M(x,y)** Return the normalized product of floating-point value \( x \) and floating-point value \( y \), having unbounded range and precision.
- **p** The intermediate product having unbounded range and precision.
- **v** The intermediate result having unbounded range and precision.
VSX Vector Round to Double-Precision Integer using round to Nearest Away XX2-form

\texttt{xvrdpi \ XT,\ XB}

\begin{verbatim}
if MSR.VSX=0 then VSX_Unavailable()

\textit{ex\_flag} \leftarrow 0b0
do \ i = 0 to 1
    \textit{reset\_xflags()}
    \textit{src} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP64(VSR[32\times BX+B].dword[i])}
    \textit{rnd} \leftarrow \text{bfp\_ROUND\_TO\_INTEGER\_NEAR\_AWAY(src)}
    \textit{vresult.dword[i]} \leftarrow \text{bfp64\_CONVERT\_FROM\_BFP(rnd)}
    if \textit{vxsnan\_flag}=1 then SetFX(FPSCR.VXSNAN)
    \textit{ex\_flag} \leftarrow \text{ex\_flag | (FPSCR.VE \& vxsnan\_flag)}
end

if ex\_flag=0 then \textit{VSR[32\times TX+T]} \leftarrow \textit{vresult}

\textbf{Let} \textit{XT} be the value \texttt{32\times TX + T}.
\textbf{Let} \textit{XB} be the value \texttt{32\times BX + B}.
\end{verbatim}

For each integer value \(i\) from 0 to 1, do the following.

\textbf{Let} \textit{src} be the double-precision floating-point operand in doubleword element \(i\) of \textit{VSR[XB]}.

\textit{src} is rounded to an integer using the rounding mode Round to Nearest Away.

The result is placed into doubleword element \(i\) of \textit{VSR[XT]} in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to \textit{VSR[XT]}.

\textbf{Special Registers Altered}

\begin{verbatim}
FX VXSNAN
\end{verbatim}

\textbf{VSR Data Layout for xvrdpi}

\begin{verbatim}
src | \textit{VSR[XB].dword[0]} | \textit{VSR[XB].dword[1]}
tgt | \textit{VSR[XT].dword[0]} | \textit{VSR[XT].dword[1]}
\end{verbatim}
VSX Vector Round to Double-Precision Integer Exact using Current rounding mode XX2-form

**xvrdpic** \(\rightarrow\) \(\rightarrow\)

Let \(XT\) be the value \(32\times TX + T\).
Let \(XB\) be the value \(32\times BX + B\).

For each integer value \(i\) from 0 to 1, do the following.

Let \(src\) be the double-precision floating-point operand in doubleword element \(i\) of \(VSR[XB]\).

\(src\) is rounded to an integer using the rounding mode specified by \(RN\).

The result is placed into doubleword element \(i\) of \(VSR[XT]\) in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to \(VSR[XT]\).

**Special Registers Altered**

\(FX\) \(XX\) \(VXSNAN\)

---

**VSR Data Layout for xvrdpic**

<table>
<thead>
<tr>
<th>src</th>
<th>(VSR[XB].dword[0])</th>
<th>(VSR[XB].dword[1])</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>(VSR[XT].dword[0])</td>
<td>(VSR[XT].dword[1])</td>
</tr>
</tbody>
</table>
VSX Vector Round to Double-Precision Integer using round toward -Infinity XX2-form

\texttt{xvrdpim} \ XT, XB

\begin{verbatim}
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0
do i = 0 to 1
   reset_xflags()
   
   src ← bfp\_CONVERT\_FROM\_BFP64(VSR[32\times BX+B].dword[i])
   rnd ← bfp\_ROUND\_TO\_INTEGER\_FLOOR(src)
   
   vresult.dword[i] ← bfp64\_CONVERT\_FROM\_BFP(rnd)
   
   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   
   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32\times TX+T] ← vresult
\end{verbatim}

Let XT be the value \(32 \times TX + T\).
Let XB be the value \(32 \times BX + B\).

For each integer value \(i\) from 0 to 1, do the following.
Let src be the double-precision floating-point operand in doubleword element \(i\) of VSR[XB].

src is rounded to an integer using the rounding mode Round toward -Infinity.

The result is placed into doubleword element \(i\) of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

\textbf{Special Registers Altered}

\texttt{FX VXSNAN}

\textbf{VSR Data Layout for xvrdpim}

\begin{verbatim}
src              VSR[XB].dword[0]              VSR[XB].dword[1]
tgt              VSR[XT].dword[0]              VSR[XT].dword[1]
\end{verbatim}
### VSX Vector Round to Double-Precision Integer using round toward +Infinity XX2-form

**xvrdpip**

**XT, XB**

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>( \text{III} )</th>
<th>B</th>
<th>233</th>
<th>B(0)</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

**ex_flag** ← 0b0
do i = 0 to 1
reset_xflags()

src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])

rnd ← bfp_ROUND_TO_INTEGER_CEIL(src)

vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

**ex_flag** ← **ex_flag** | (FPSCR.VE & vxsnan_flag)
end

if **ex_flag**=0 then VSR[32×TX+T] ← vresult

Let **XT** be the value \(32×TX + T\).

Let **XB** be the value \(32×BX + B\).

For each integer value \(i\) from 0 to 1, do the following.

Let **src** be the double-precision floating-point operand in doubleword element \(i\) of VSR[**XB**].

**src** is rounded to an integer using the rounding mode Round toward +Infinity.

The result is placed into doubleword element \(i\) of VSR[**XT**] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[**XT**].

Special Registers Altered

**FX VXSNAN**

### VSX Vector Round to Double-Precision Integer using round toward Zero XX2-form

**xvrdpiz**

**XT, XB**

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>( \text{III} )</th>
<th>B</th>
<th>217</th>
<th>B(0)</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

**ex_flag** ← 0b0
do i = 0 to 1
reset_xflags()

src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])

rnd ← bfp_ROUND_TO_INTEGER_TRUNC(src)

vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

**ex_flag** ← **ex_flag** | (FPSCR.VE & vxsnan_flag)
end

if **ex_flag**=0 then VSR[32×TX+T] ← vresult

Let **XT** be the value \(32×TX + T\).

Let **XB** be the value \(32×BX + B\).

For each integer value \(i\) from 0 to 1, do the following.

Let **src** be the double-precision floating-point operand in doubleword element \(i\) of VSR[**XB**].

**src** is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into doubleword element \(i\) of VSR[**XT**] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[**XT**].

Special Registers Altered

**FX VXSNAN**
VSX Vector Reciprocal Estimate
Double-Precision XX2-form

Let $XT$ be the value $32 \times TX + T$.
Let $XB$ be the value $32 \times BX + B$.

For each integer value $i$ from 0 to 1, do the following.

Let $src$ be the double-precision floating-point operand in doubleword element $i$ of VSR[$XB$].

A double-precision floating-point estimate of the reciprocal of $src$ is placed into doubleword element $i$ of VSR[$XT$] in double-precision format.

Unless the reciprocal of $src$ would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of $src$. That is,

$$\left| \frac{1}{src} - \text{estimate} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>−Infinity</td>
<td>−Zero</td>
<td>None</td>
</tr>
<tr>
<td>−Zero</td>
<td>−Infinity</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity</td>
<td>ZX</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if ZE=1.
2. No result if VE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[$XT$].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FX, OX, UX, ZX, VXSNAN

VSR Data Layout for xvredp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[$XB$].dword[0]</th>
<th>VSR[$XB$].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[$XT$].dword[0]</td>
<td>VSR[$XT$].dword[1]</td>
</tr>
</tbody>
</table>
VSX Vector Reciprocal Estimate
Single-Precision XX2-form

Let XT be the value $32 \times TX + T$.
Let XB be the value $32 \times XB + B$.

For each integer value $i$ from 0 to 3, do the following.
Let src be the single-precision floating-point operand in word element $i$ of VSR[XB].

A single-precision floating-point estimate of the reciprocal of src is placed into word element $i$ of VSR[XT] in single-precision format.

Unless the reciprocal of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{src}}{\frac{1}{src}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>None</td>
</tr>
<tr>
<td>$-\infty$</td>
<td>None</td>
<td>ZX</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$+\infty$</td>
<td>None</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>None</td>
<td>ZX</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>VXSNAN</td>
</tr>
</tbody>
</table>

1. No result if ZE=1.
2. No result if VE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FX  OX  UX  ZX  VXSNAN

VSR Data Layout for xvresp

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[XB].word[0]</td>
<td>VSR[XT].word[0]</td>
</tr>
</tbody>
</table>

0 32 64 96 127
VSX Vector Round to Single-Precision Integer using round to Nearest Away XX2-form

**xvrspi** XT, XB

<table>
<thead>
<tr>
<th>i</th>
<th>60</th>
<th>T</th>
<th>#/</th>
<th>B</th>
<th>137</th>
<th>B/T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>30</td>
</tr>
</tbody>
</table>

- If MSR.VSX=0 then VSXUnavailable()
- \[\text{ex\_flag} \leftarrow \text{0b0}\]
- \[\text{do } i = 0 \text{ to } 3 \]
  - \[\text{reset\_flags()}\]
  - \[\text{src} \leftarrow \text{bfp\_CONVERT\_FROM\_BFP32(VSR[32\times BX+B].word[i])}\]
  - \[\text{rnd} \leftarrow \text{bfp\_ROUND\_TO\_INTEGER\_NEAR\_AWAY(src)}\]
  - \[\text{vresult\_word[i]} \leftarrow \text{bfp32\_CONVERT\_FROM\_BFP(rnd)}\]
  - \[\text{if vxsnan\_flag=1 then SetFX(FPSCR.VXSNAN)}\]
  - \[\text{ex\_flag} \leftarrow \text{ex\_flag | (FPSCR.VE \& vxsnan\_flag)}\]
- \[\text{end}\]
- \[\text{if ex\_flag=0 then VSR[32\times TX+T] } \leftarrow \text{vresult}\]

**VSR Data Layout for xvrspi**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>30</td>
</tr>
</tbody>
</table>

Let XT be the value 32 × TX + T.
Let XB be the value 32 × BX + B.

For each integer value i from 0 to 3, do the following.
Let \(\text{src}\) be the single-precision floating-point operand in word element i of \(\text{VSR[XB]}\).
\(\text{src}\) is rounded to an integer using the rounding mode Round to Nearest Away.
The result is placed into word element i of \(\text{VSR[XT]}\) in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to \(\text{VSR[XT]}\).

**Special Registers Altered**

FX VXSNAN
VSX Vector Round to Single-Precision Integer
Exact using Current rounding mode XX2-form

Let XT be the value $32 \times T + T$.
Let XB be the value $32 \times B + B$.

For each integer value i from 0 to 3, do the following.
Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer value using the rounding mode specified by RN.

The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
FX  XX  VXSNAN

VSR Data Layout for xvrspic

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>171</td>
<td>60</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>
VSX Vector Round to Single-Precision Integer using round toward -Infinity XX2-form

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 3, do the following.

Let \( src \) be the single-precision floating-point operand in word element \( i \) of \( VSR[XB] \).

\( src \) is rounded to an integer using the rounding mode Round toward -Infinity.

The result is placed into word element \( i \) of \( VSR[XT] \) in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( VSR[XT] \).

**Special Registers Altered**

- FX VXSNAN

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 3, do the following.

Let \( src \) be the single-precision floating-point operand in word element \( i \) of \( VSR[XB] \).

\( src \) is rounded to an integer using the rounding mode Round toward -Infinity.

The result is placed into word element \( i \) of \( VSR[XT] \) in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( VSR[XT] \).

**Special Registers Altered**

- FX VXSNAN
### VSX Vector Round to Single-Precision Integer using round toward +Infinity XX2-form

**xvrspip**

**XT, XB**

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>169</th>
<th>64</th>
</tr>
</thead>
</table>

If MSR.VSX=0 then VSX_Unavailable()

```plaintext
ex_flag ← 0b0
do i = 0 to 3
    reset_xflags()
    src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
    rnd ← bfp_ROUND_TO_INTEGER_CEIL(src)
    vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)
endif
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end
if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let **XT** be the value $32\times TX + T$.
Let **XB** be the value $32\times BX + B$.

For each integer value $i$ from 0 to 3, do the following.

Let **src** be the single-precision floating-point operand in word element $i$ of VSR[XB].

**src** is rounded to an integer using the rounding mode Round toward +Infinity.

The result is placed into word element $i$ of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

### Special Registers Altered

- **FX**
- **VXSNAN**

### VSR Data Layout for xvrspip

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
</tr>
</thead>
</table>

### VSX Vector Round to Single-Precision Integer using round toward Zero XX2-form

**xvrspiz**

**XT, XB**

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>153</th>
<th>64</th>
</tr>
</thead>
</table>

If MSR.VSX=0 then VSX_Unavailable()

```plaintext
ex_flag ← 0b0
do i = 0 to 3
    reset_xflags()
    src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
    rnd ← bfp_ROUND_TO_INTEGER_TRUNC(src)
    vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)
endif
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end
if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let **XT** be the value $32\times TX + T$.
Let **XB** be the value $32\times BX + B$.

For each integer value $i$ from 0 to 3, do the following.

Let **src** be the single-precision floating-point operand in word element $i$ of VSR[XB].

**src** is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into word element $i$ of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

### Special Registers Altered

- **FX**
- **VXSNAN**
Let $XT$ be the value $32 \times TX + T$.
Let $XB$ be the value $32 \times BX + B$.

For each integer value $i$ from 0 to 1, do the following.
Let $src$ be the double-precision floating-point operand in doubleword element $i$ of $VSR[XB]$.

A double-precision floating-point estimate of the reciprocal square root of $src$ is placed into doubleword element $i$ of $VSR[XT]$ in double-precision format.

Unless the reciprocal of the square root of $src$ would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of $src$. That is,

\[
\frac{|estimate - \frac{1}{\sqrt{src}}|}{\frac{1}{\sqrt{src}}} \leq \frac{1}{16384}
\]

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>−Infinity</td>
<td>QNaN$^1$</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>−Finite</td>
<td>QNaN$^1$</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>−Zero</td>
<td>−Infinity$^2$</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity$^2$</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN$^1$</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if VE=1.
2. No result if ZE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to $VSR[XT]$.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

**Special Registers Altered**

$FX$ $ZX$ $VXSNAN$ $VXSQRT$
VSX Vector Reciprocal Square Root Estimate
Single-Precision XX2-form

xvrsqrtesp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>138</th>
<th>16384</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
    reset_xflags()
    src ← bfp_CONVERT_FROM_BFP32(VSR[BX+B].word[i])
    v ← bfp_RECIPROCAL_SQUARE_ROOT_ESTIMATE(src)
    rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
    vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)
    if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
    if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
    if zx_flag=1 then SetFX(FPSCR.ZX)
    ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
    ex_flag ← ex_flag | (FPSCR.VE & vxsqrt_flag)
    ex_flag ← ex_flag | (FPSCR.ZE & zx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.
Let src be the single-precision floating-point operand in word element i of VSR[XB].

A single-precision floating-point estimate of the reciprocal square root of src is placed into word element i of VSR[XT] in single-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

\[
\left| \frac{1}{\sqrt{src}} - \text{estimate} \right| \leq \frac{1}{16384}
\]

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>–Zero</td>
<td>–Infinity(^2)</td>
<td>ZX</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN (\neq)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN (\neq)</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if VE=1.
2. No result if ZE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered
FX ZX VXSNAN VXSQRT

VSR Data Layout for xvrsqrtesp

|-----|-----------------|-----------------|-----------------|-----------------|

Chapter 7. Vector-Scalar Extension Facility 947
VSX Vector Square Root Double-Precision

**XX2-form**

### xvsqrtdp

<table>
<thead>
<tr>
<th>XT, XB</th>
<th>0</th>
<th>6</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>203</th>
<th>B[11:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
  reset_xflags()

  src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
  v ← bfp_SQUARE_ROOT(src)
  rnd ← bfp_ROUND_TO_BFP64(0b0, FPSCR.RN, v)

  vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
if xx_flag=1 then SetFX(FPSCR.XX)

ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
ex_flag ← ex_flag | (FPSCR.VE & vxsqrt_flag)
ex_flag ← ex_flag | (FPSCR.XE & xx_flag)

end

if ex_flag=0 then VSR[32×TX+T] ← vresult

#### Let

- XT be the value 32×TX + T.
- XB be the value 32×BX + B.

For each integer value \(i\) from 0 to 1, do the following.

Let \(src\) be the double-precision floating-point operand in doubleword element \(i\) of VSR[XB].

The unbounded-precision square root of \(src\) is produced.

See Table 141.

The intermediate result is rounded to double-precision using the rounding mode specified by \(RN\).


The result is placed into doubleword element \(i\) of VSR[XT] in double-precision format.

See Table 114, “Vector Floating-Point Final Result,” on page 823.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

### Special Registers Altered

- FX
- XX
- VXSNAN
- VXSQRT

#### VSR Data Layout for xvsqrtdp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>VSR[XB].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← SQRT(src)</td>
<td>v ← +Infinity</td>
<td>v ← src</td>
<td>v ← Q(src)</td>
</tr>
<tr>
<td>vxsnan_flag ← 1</td>
<td>vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxsnan_flag ← 1</td>
<td></td>
</tr>
</tbody>
</table>

Explanation:

- src: The double-precision floating-point value in doubleword element \(i\) of VSR[XB] (where \(i=0,1\)).
- dQNaN: Default quiet NaN (0x7FF8_0000_0000_0000).
- NZF: Nonzero finite number.
- SQRT(x): The unbounded-precision square root of the floating-point value \(x\).
- Q(x): Return a QNaN with the payload of \(x\).
- v: The intermediate result having unbounded significand precision and unbounded exponent range.

Table 141. Actions for xvsqrtdp
Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.
Let src be the single-precision floating-point operand in word element i of VSR[XB].

The unbounded-precision square root of src is produced.

See Table 142.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.


The result is placed into word element i of VSR[XT] in single-precision format.

See Table 114, “Vector Floating-Point Final Result,” on page 823.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

### Special Registers Altered

FX XX VXSNAN VXSQRT

---

### Table 142. Actions for xvsqrtsp

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>-NZF</td>
<td>-Zero</td>
<td>+Zero</td>
<td>+NZF</td>
<td>+Infinity</td>
<td>QNaN</td>
<td>SNaN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← SQRT(src)</td>
<td>v ← +Infinity</td>
<td>v ← src</td>
<td>v ← Q(src)</td>
<td>v ← vsqrt_flag</td>
<td>v ← +Nan</td>
</tr>
<tr>
<td>vsqrt_flag ← 1</td>
<td>vsqrt_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation:
- src: The single-precision floating-point value in word element i of VSR[XB] (where i=0,1,2,3).
- dQNaN: Default quiet NaN (0x7FC0_0000).
- NZF: Nonzero finite number.
- SQRT(x): The unbounded-precision square root of the floating-point value x.
- Q(x): Return a QNaN with the payload of x.
- v: The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Vector Subtract Double-Precision
XX3-form

\[
\text{vsubdp} \quad \text{XT,XA,XB}
\]

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XA \) be the value \( 32 \times AX + A \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each integer value \( i \) from 0 to 1, do the following.

Let \( \text{src1} \) be the double-precision floating-point operand in doubleword element \( i \) of \( \text{VSR}[XA] \).

Let \( \text{src2} \) be the double-precision floating-point operand in doubleword element \( i \) of \( \text{VSR}[XB] \).

\( \text{src2} \) is negated and added\(^1\) to \( \text{src1} \), producing a sum having unbounded range and precision.

The sum is normalized\(^2\).

See Table 143.

The intermediate result is rounded to double-precision using the rounding mode specified by \( RN \).


The result is placed into doubleword element \( i \) of \( \text{VSR}[XT] \) in double-precision format.

See Table 114, “Vector Floating-Point Final Result,” on page 823.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( \text{VSR}[XT] \).

**Special Registers Altered**

\( FX \quad OX \quad UX \quad XX \quad VXSNAN \quad VXSI \)

### VSR Data Layout for vsubdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].dword[0]</td>
<td>VSR[XB].dword[1]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

---

950  Power ISA™ I
### Table 143. Actions for `xvsubdp`

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1, src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← S(src1, src2)</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← –src2</td>
<td>v ← –Zero</td>
<td>v ← Rezd</td>
<td>v ← –src2</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← –src2</td>
<td>v ← Rezd</td>
<td>v ← +Zero</td>
<td>v ← –src2</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1, src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← S(src1, src2)</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

**Explanation:**

- `src1` The double-precision floating-point value in doubleword element i of VSR[XA] (where i={0,1}).
- `src2` The double-precision floating-point value in doubleword element i of VSR[XB] (where i={0,1}).
- `dQNaN` Default quiet NaN (0x7FF8_0000_0000_0000).
- `NZF` Nonzero finite number.
- `Rezd` Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- `S(x,y)` Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
  - **Note:** If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- `Q(x)` Return a QNaN with the payload of x.
- `v` The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Vector Subtract Single-Precision
XX3-form

Let $XT$ be the value $32\times TX + T$.
Let $XA$ be the value $32\times AX + A$.
Let $XB$ be the value $32\times BX + B$.

For each integer value $i$ from 0 to 3, do the following.
Let $src1$ be the single-precision floating-point operand in word element $i$ of $VSR[XA]$.
Let $src2$ be the single-precision floating-point operand in word element $i$ of $VSR[XB]$.

$src2$ is negated and added\(^1\) to $src1$, producing a sum having unbounded range and precision.

The sum is normalized\(^2\).

See Table 144.

The intermediate result is rounded to single-precision using the rounding mode specified by $RN$.


The result is placed into word element $i$ of $VSR[XT]$ in single-precision format.

See Table 114, “Vector Floating-Point Final Result,” on page 823.

If a trap-enabled exception occurs in any element of the vector, no results are written to $VSR[XT]$.

Special Registers Altered

<table>
<thead>
<tr>
<th>FX</th>
<th>OX</th>
<th>UX</th>
<th>XX</th>
<th>VXSNAN</th>
<th>VXISI</th>
</tr>
</thead>
</table>

VSR Data Layout for xvsubsp

|------|----------------|----------------|----------------|----------------|

---

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 144. Actions for xvsubsp

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1, src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← S(src1, src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← -Zero</td>
<td>v ← Rezd</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← Rezd</td>
<td>v ← +Zero</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1, src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← S(src1, src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>vxsnan_flag ← 1</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

**Explanation:**
- src1: The single-precision floating-point value in word element i of VSR[XA] (where i=0,1,2,3).
- src2: The single-precision floating-point value in word element i of VSR[XB] (where i=0,1,2,3).
- dQNaN: Default quiet NaN (0x7FC0_0000).
- NZF: Nonzero finite number.
- Rezd: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- S(x,y): Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
- Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- Q(x): Return a QNaN with the payload of x.
- v: The intermediate result having unbounded significand precision and unbounded exponent range.
Version 3.1

VSX Vector Test for software Divide
Double-Precision XX3-form

xvtdivdp BF,XA,XB

if MSR.VSX=0 then VSX_Unavailable();

eq_flag ← 0b0
gt_flag ← 0b0

do i = 0 to 1
src1 ← VSR[32×AX+A].dword[i]
src2 ← VSR[32×BX+B].dword[i]
e_a ← src1.bit[1:11] - 1023
e_b ← src2.bit[1:11] - 1023
fe_flag ← fe_flag | IsNaN(src1) | IsInf(src1) |
| IsNaN(src2) | IsInf(src2) | IsZero(src2) |
| e_b <= -1022 ) | |
| e_b >= 1021 ) | |
| !IsZero(src1) & ( | e_a - e_b | >= 1023 ) ) | |
| !IsZero(src1) & ( | e_a - e_b | <= -1021 ) ) | |
| !IsZero(src1) & ( e_a <= -970 ) |
fg_flag ← fg_flag | IsInf(src1) | IsInf(src2) |
| IsZero(src2) | IsDen(src2) |
end

fl_flag ← xvredp_error() <= 2⁻¹⁴
CR[BF] ← 0b1 || fg_flag || fe_flag || 0b0

LetXA be the value 32×AX + A.
LetXB be the value 32×BX + B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each integer value i from 0 to 1, do the following.
Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

Let e_a be the unbiased exponent of src1.
Let e_b be the unbiased exponent of src2.

fe_flag is set to 1 for any of the following conditions.
– src1 is a NaN or an infinity.
– src2 is a zero, a NaN, or an infinity.
– e_b is less than or equal to -1022.
– e_b is greater than or equal to 1021.
– src1 is not a zero and the difference, e_a - e_b, is greater than or equal to 1023.
– src1 is not a zero and the difference, e_a - e_b, is less than or equal to -1021.
– src1 is not a zero and e_a is less than or equal to -970.

fg_flag is set to 1 for any of the following conditions.
– src1 is an infinity.
– src2 is a zero, an infinity, or a denormalized value.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered
CR field BF

VSR Data Layout for xvtdivdp

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].dword[0]</th>
<th>VSR[XA].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2</th>
<th>VSR[XB].dword[0]</th>
<th>VSR[XB].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### VSX Vector Test for software Divide
### Single-Precision XX3-form

**xvtdivsp** BF,XA,XB

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>BF</th>
<th>A</th>
<th>B</th>
<th>93</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VSX=0 then VSX_Unavailable()

eq_flag ← 0b0
gt_flag ← 0b0

do i = 0 to 3
    src1 ← VSR[32×AX+A].word[i]
    src2 ← VSR[32×BX+B].word[i]
    e_a ← src1.bit[1:8] - 127
    e_b ← src2.bit[1:8] - 127
    fe_flag ← fe_flag | IsNaN(src1) | IsInf(src1) | IsNaN(src2) | IsInf(src2) | IsZero(src2) | ( e_b <= -126 ) | ( e_b >= 125 ) | ( IsZero(src1) & ( e_a - e_b ) >= 127 ) | ( IsZero(src1) & ( e_a - e_b ) <= -125 ) | ( e_a <= -103 )
    fg_flag ← fg_flag | IsInf(src1) | IsZero(src2) | IsDen(src2)
end

fl_flag ← xvredp_error() <= 2-14
CR.field[BF] ← 0b1 || fg_flag || fe_flag || 0b0

Let **XA** be the value 32×AX + A.
Let **XB** be the value 32×BX + B.

**fe_flag** is initialized to 0.
**fg_flag** is initialized to 0.

For each integer value **i** from 0 to 3, do the following.
Let **src1** be the single-precision floating-point operand in word element **i** of VSR[**XA**].
Let **src2** be the single-precision floating-point operand in word element **i** of VSR[**XB**].

Let **e_a** be the unbiased exponent of **src1**.
Let **e_b** be the unbiased exponent of **src2**.

**fe_flag** is set to 1 for any of the following conditions.
- **src1** is a NaN or an infinity.
- **src2** is a zero, a NaN, or an infinity.
- **e_b** is less than or equal to -126.
- **e_b** is greater than or equal to 125.
- **src1** is not a zero and the difference, **e_a - e_b**, is greater than or equal to 127.
- **src1** is not a zero and the difference, **e_a - e_b**, is less than or equal to -125.
- **src1** is not a zero and **e_a** is less than or equal to -103.

**fg_flag** is set to 1 for any of the following conditions.
- **src1** is an infinity.
- **src2** is a zero, an infinity, or a denormalized value.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

**Special Registers Altered**
CR field BF

### VSR Data Layout for xvtdivsp

|-------|-------------------|-------------------|-------------------|-------------------|

<table>
<thead>
<tr>
<th>0</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>127</th>
</tr>
</thead>
</table>

VSX Vector Test for software Square Root
Double-Precision XX2-form

xvtsqrtdp  BF,XB

if MSR.VSX=0 then VSX_Unavailable();
fe_flag ← 0b0
fg_flag ← 0b0
do i = 0 to 1
  src ← VSR[32xBX+B].dword[i]
  e_b ← src.bit[1:11] - 1023
  fe_flag ← fe_flag | IsNaN(src) | IsInf(src) | IsZero(src) | IsNeg(src) | ( e_b <= -970 )
  fg_flag ← fg_flag | IsInf(src) | IsZero(src) | IsDen(src)
end
fl_flag = xvrsqrtedp_error() <= 2^-14
CR.field[BF] ← 0b1 || fg_flag || fe_flag || 0b0

Let XB be the value 32xBX + B.
fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each integer value i from 0 to 1, do the following.
Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].
Let e_b be the unbiased exponent of src.
fe_flag is set to 1 for any of the following conditions.
  - src is a zero, a NaN, an infinity, or a negative value.
  - e_b is less than or equal to -970.
fg_flag is set to 1 for the following condition.
  - src is a zero, an infinity, or a denormalized value.

CR field BF is set to the value
0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered
CR field BF

VSX Vector Test for software Square Root
Single-Precision XX2-form

xvtsqrtsp  BF,XB

if MSR.VSX=0 then VSX_Unavailable();
fe_flag ← 0b0
fg_flag ← 0b0
do i = 0 to 3
  src ← VSR[32xBX+B].word[i]
  e_b ← src.bit[1:8] - 127
  fe_flag ← fe_flag | IsNaN(src) | IsInf(src) | IsZero(src) | IsNeg(src) | ( e_b <= -103 )
  fg_flag ← fg_flag | IsInf(src) | IsZero(src) | IsDen(src)
end
fl_flag = xvrsqrtesp_error() <= 2^-14
CR.field[BF] ← 0b1 || fg_flag || fe_flag || 0b0

Let XB be the value 32xBX + B.
fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each integer value i from 0 to 3, do the following.
Let src be the single-precision floating-point operand in word element i of VSR[XB].
Let e_b be the unbiased exponent of src.
fe_flag is set to 1 for any of the following conditions.
  - src is a zero, a NaN, an infinity, or a negative value.
  - e_b is less than or equal to -103.
fg_flag is set to 1 for the following condition.
  - src is a zero, an infinity, or a denormalized value.

CR field BF is set to the value
0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered
CR field BF
Chapter 7. Vector-Scalar Extension Facility

VSX Vector Test Data Class Double-Precision
XX2-form

Let XB be the sum 32×BX + B.
Let XT be the sum 32×TX + T.
Let DCMX be the value dc concatenated with dm concatenated with dx.

For each integer value i from 0 to 1, do the following.
Let src be the double-precision floating-point value in doubleword element i of VSR[XB].

If src matches one of the 7 possible data classes specified by DCMX (Data Class Mask), the contents of doubleword element i of VSR[XT] are set to 0xFFFF_FFFF_FFFF_FFFF. Otherwise, the contents of doubleword element i of VSR[XT] are set to 0x0000_0000_0000_0000.

<table>
<thead>
<tr>
<th>DCMX bit</th>
<th>Data Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>+Infinity</td>
</tr>
<tr>
<td>2</td>
<td>-Infinity</td>
</tr>
<tr>
<td>3</td>
<td>+Zero</td>
</tr>
<tr>
<td>4</td>
<td>-Zero</td>
</tr>
<tr>
<td>5</td>
<td>+Denormal</td>
</tr>
<tr>
<td>6</td>
<td>-Denormal</td>
</tr>
</tbody>
</table>

Special Registers Altered:
None
VSX Vector Test Data Class Single-Precision
XX2-form

xvtstdcsp XT, XB, DCMX

if MSR.VSX=0 then VSX_Unavailable()

DCMX ← dc || dm || dx
do i = 0 to 3
src ← VSR[32×BX+B].word[i]
sign ← src.bit[0]
exponent ← src.bit[1:8]
fraction ← src.bit[9:31]
class.Infinity ← (exponent = 0xFF) & (fraction = 0)
class.NaN ← (exponent = 0xFF) & (fraction != 0)
class.Zero ← (exponent = 0x00) & (fraction = 0)
class.Denormal ← (exponent = 0x00) & (fraction != 0)
match ← (DCMX.bit[0] & class.NaN)              |
if match = 1 then
  VSR[32×TX+T].dword[i] ← 0xFFFF_FFFF
else
  VSR[32×TX+T].dword[i] ← 0x0000_0000
end

Let XB be the sum 32×BX + B.
Let XT be the sum 32×TX + T.
Let DCMX be the value dc concatenated with dm concatenated with dx.

For each integer value i from 0 to 3, do the following.
Let src be the single-precision floating-point value in word element i of VSR[XB].

If src matches one of the 7 possible data classes specified by DCMX (Data Class Mask), the contents of word element i of VSR[XT] are set to 0xFFFF_FFFF. Otherwise, the contents of word element i of VSR[XT] are set to 0x0000_0000.

DCMX bit Data Class
0     NaN
1     +Infinity
2     -Infinity
3     +Zero
4     -Zero
5     +Denormal
6     -Denormal

Special Registers Altered:
None

VSR Data Layout for xvtstdcsp

|-----|----------------|----------------|----------------|----------------|
VSX Vector Test Least-Significant Bit by Byte
XX2-form

```
xvtlsbb BF,XB
```

```
if MSR.VSX=0 then VSX_Unavailable()

ALL_TRUE ← 1
ALL_FALSE ← 1

do i = 0 to 15
   ALL_TRUE ← ALL_TRUE & (VSR[XB].byte[i].bit[7]=1)
   ALL_FALSE ← ALL_FALSE & (VSR[XB].byte[i].bit[7]=0)
endo

CR.field_WRITE(BF, ALL_TRUE, 0, ALL_FALSE, 0);
```

Set CR field BF to indicate if bit 7 of every byte element in VSR[XB] is equal to 1 (ALL_TRUE) or equal to 0 (ALL_FALSE).

**Special Registers Altered**
- CR field BF

**VSR Data Layout for xvtlsbb**

|-----|-----------------|-----------------|-----------------|-----------------|

<table>
<thead>
<tr>
<th>0</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>127</th>
</tr>
</thead>
</table>

**Programming Note**

This instruction following any Vector Compare provides the ability to direct the summary status of the Vector Compare to any CR field, not just CR field 6 when Rc=1.
VSX Vector Extract Exponent
Double-Precision XX2-form

\[
xvxexpdp \quad XT,XB
\]

\[
\begin{array}{cccccc}
0 & 6 & T & 0 & B & 475 \\
60 & 6 & 11 & 16 & 21 & 30
\end{array}
\]

if MSR.VSX=0 then VSX_Unavailable();

do i = 0 to 1
\[
src \leftarrow VSR[32\times BX+B].dword[i]
\]
\[
VSR[32\times TX+T].dword[i] \leftarrow EXTZ64(src.bit[1:11])
\]
end

Let XT be the sum \(32\times TX + T\).
Let XB be the sum \(32\times BX + B\).

For each integer value \(i\) from 0 to 1, do the following.
Let \(src\) be the double-precision floating-point value in doubleword element \(i\) of VSR[XB].

The value of the exponent field in \(src\) is placed into doubleword element \(i\) of VSR[XT] in unsigned integer format.

Special Registers Altered:
None

VSR Data Layout for \(xvxexpdp\)

\[
\begin{array}{cc}
src & VSR[XB].dword[0] \\
& VSR[XB].dword[1] \\
tgt & VSR[XT].dword[0] \\
& VSR[XT].dword[1]
\end{array}
\]

VSX Vector Extract Exponent Single-Precision
XX2-form

\[
xvxexpsp \quad XT,XB
\]

\[
\begin{array}{cccccc}
0 & 6 & T & 8 & B & 475 \\
60 & 6 & 11 & 16 & 21 & 30
\end{array}
\]

if MSR.VSX=0 then VSX_Unavailable();

do i = 0 to 3
\[
src \leftarrow VSR[32\times BX+B].word[i]
\]
\[
VSR[32\times TX+T].word[i] \leftarrow EXTZ32(src.bit[1:8])
\]
end

Let XT be the sum \(32\times TX + T\).
Let XB be the sum \(32\times BX + B\).

For each integer value \(i\) from 0 to 3, do the following.
Let \(src\) be the single-precision floating-point value in word element \(i\) of VSR[XB].

The value of the exponent field in \(src\) is placed into word element \(i\) of VSR[XT] in unsigned integer format.

Special Registers Altered:
None

VSR Data Layout for \(xvxexpsp\)

\[
\begin{array}{cccc}
\end{array}
\]
Chapter 7. Vector-Scalar Extension Facility

VSX Vector Extract Significand
Double-Precision XX2-form

vt vsagydp  (XT,XB)

<table>
<thead>
<tr>
<th>i</th>
<th>60</th>
<th>56</th>
<th>52</th>
<th>48</th>
<th>44</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>1</td>
<td>B</td>
<td>475</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

for i = 0 to 1

src ← VSR[32×BX+B].dword[i]

exponent ← EXTZ(src.bit[1:11])

t fraction ← EXTZ64(src.bit[12:63])

if (exponent != 0) & (exponent != 2047) then

t fraction ← fraction | 0x0010_0000_0000_0000

VSR[32×TX+T].dword[i] ← fraction

der

Let XT be the sum 32×TX + T.

Let XB be the sum 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point value in doubleword element i of VSR[XB].

The significand of src is placed into doubleword element i of VSR[XT] in unsigned integer format. If src is a normal value, the implicit leading bit is set to 1.

Special Registers Altered:
None

VSR Data Layout for xvxsigdp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>VSR[XB].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>

VSR Data Layout for xvxsigsp

|-----|----------------|----------------|----------------|----------------|

VSX Vector Extract Significand
Single-Precision XX2-form

vt vsagyds  (XT,XB)

<table>
<thead>
<tr>
<th>i</th>
<th>60</th>
<th>56</th>
<th>48</th>
<th>44</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>9</td>
<td>B</td>
<td>475</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

for i = 0 to 3

src ← VSR[32×BX+B].word[i]

exponent ← EXTZ(src.bit[1:8])

t fraction ← EXTZ32(src.bit[9:31])

if (exponent != 0) & (exponent != 255) then

t fraction ← fraction | 0x0080_0000

VSR[32×TX+T].word[i] ← fraction

der

Let XT be the sum 32×TX + T.

Let XB be the sum 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point value in word element i of VSR[XB].

The significand of src is placed into word element i of VSR[XT] in unsigned integer format. If src is a normal value, the implicit leading bit is set to 1.

Special Registers Altered:
None
VSX Vector Blend Variable Byte 8RR:XX4-form

`xxblendvb` XT,XA,XB,XC

Prefix:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>//</th>
<th>///</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>8</td>
<td>12</td>
<td>14</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th></th>
<th>33</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>8</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
<tr>
<td>1</td>
<td>35</td>
<td>36</td>
<td>38</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 15
  if MSR[32×CX+C].byte[i].bit[0]=0 then
    MSR[32×TX+T].byte[i] ← MSR[32×AX+A].byte[i]
  else
    MSR[32×TX+T].byte[i] ← MSR[32×BX+B].byte[i]
end

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XC be the value 32×CX + C.

For each integer value i from 0 to 15, do the following.
  If the contents of bit 0 of byte element i of MSR[XC] is equal to 0, the contents of byte element i of MSR[XA] are placed into byte element i of MSR[XT]. Otherwise, the contents of byte element i of MSR[XB] are placed into byte element i of MSR[XT].

Special Registers Altered:
None

VSX Vector Blend Variable Halfword 8RR:XX4-form

`xxblendvh` XT,XA,XB,XC

Prefix:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>//</th>
<th>///</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>8</td>
<td>12</td>
<td>14</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th></th>
<th>33</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>CxB.XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>8</td>
<td>16</td>
<td>21</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>35</td>
<td>36</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 7
  if MSR[32×CX+C].hword[i].bit[0]=0 then
    MSR[32×TX+T].hword[i] ← MSR[32×AX+A].hword[i]
  else
    MSR[32×TX+T].hword[i] ← MSR[32×BX+B].hword[i]
end

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XC be the value 32×CX + C.

For each integer value i from 0 to 7, do the following.
  If the contents of bit 0 of halfword element i of MSR[XC] is equal to 0, the contents of halfword element i of MSR[XA] are placed into halfword element i of MSR[XT]. Otherwise, the contents of halfword element i of MSR[XB] are placed into halfword element i of MSR[XT].

Special Registers Altered:
None

VSR Data Layout for `xxblendvb`

```
```

VSR Data Layout for `xxblendvh`

```
```
Chapter 7. Vector-Scalar Extension Facility

VSX Vector Blend Variable Word
8RR:XX4-form

**xxblendw**  XT,XA,XB,XC

**Prefix:**

```
1 1 0 // // 31
```

**Suffix:**

```
0 6 8 12 14 30 31
```

*if MSR.VSX=0 then VSX_Unavailable()*

```plaintext
do i = 0 to 3
  if VSR[32×CX+C].word[i].bit[0]=0 then
    VSR[32×TX+T].word[i] ← VSR[32×AX+A].word[i]
  else
    VSR[32×TX+T].word[i] ← VSR[32×BX+B].word[i]
  end
```

Let **XT** be the value 32×TX + T.
Let **XA** be the value 32×AX + A.
Let **XB** be the value 32×BX + B.
Let **XC** be the value 32×CX + C.

For each integer value **i** from 0 to 3, do the following.
If the contents of bit 0 of word element **i** of VSR[XC] is equal to 0, the contents of word element **i** of VSR[XA] are placed into word element **i** of VSR[XT]. Otherwise, the contents of word element **i** of VSR[XB] are placed into word element **i** of VSR[XT].

**Special Registers Altered:**
None

**VSX Data Layout for xxblendw**

```
```

VSX Vector Blend Variable Doubleword
8RR:XX4-form

**xxblendvd**  XT,XA,XB,XC

**Prefix:**

```
1 1 0 // // 31
```

**Suffix:**

```
0 6 8 12 14 30 31
```

*if MSR.VSX=0 then VSX_Unavailable()*

```plaintext
do i = 0 to 1
  if VSR[32×CX+C].dword[i].bit[0]=0 then
    VSR[32×TX+T].dword[i] ← VSR[32×AX+A].dword[i]
  else
    VSR[32×TX+T].dword[i] ← VSR[32×BX+B].dword[i]
  end
```

Let **XT** be the value 32×TX + T.
Let **XA** be the value 32×AX + A.
Let **XB** be the value 32×BX + B.
Let **XC** be the value 32×CX + C.

For each integer value **i** from 0 to 1, do the following.
If the contents of bit 0 of doubleword element **i** of VSR[XC] is equal to 0, the contents of doubleword element **i** of VSR[XA] are placed into doubleword element **i** of VSR[XT]. Otherwise, the contents of doubleword element **i** of VSR[XB] are placed into doubleword element **i** of VSR[XT].

**Special Registers Altered:**
None

**VSX Data Layout for xxblendvd**

```
src1  VSR[XA].dword[0]  VSR[XA].dword[1]
src2  VSR[XB].dword[0]  VSR[XB].dword[1]
src3  VSR[XC].dword[0]  VSR[XC].dword[1]
tgt  VSR[XT].dword[0]  VSR[XT].dword[1]
```
VSX Vector Byte-Reverse Doubleword

XX2-form

xxbrd XT,XB

| 60 | 6 | 23 | B | 475 | E0X |

if MSR.VSX=0 then VSX_Unavailable()

doi = 0 to 1

vsrc ← VSR[32×BX+B].dword[i]

do j = 0 to 7

VSR[32×TX+T].dword[i].byte[j] ← vsrc.byte[7-j]

di end

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

The contents of byte 7 of doubleword element i of VSR[XB] are placed into byte 0 of doubleword element i of VSR[XT].

The contents of byte 6 of doubleword element i of VSR[XB] are placed into byte 1 of doubleword element i of VSR[XT].

The contents of byte 5 of doubleword element i of VSR[XB] are placed into byte 2 of doubleword element i of VSR[XT].

The contents of byte 4 of doubleword element i of VSR[XB] are placed into byte 3 of doubleword element i of VSR[XT].

The contents of byte 3 of doubleword element i of VSR[XB] are placed into byte 4 of doubleword element i of VSR[XT].

The contents of byte 2 of doubleword element i of VSR[XB] are placed into byte 5 of doubleword element i of VSR[XT].

The contents of byte 1 of doubleword element i of VSR[XB] are placed into byte 6 of doubleword element i of VSR[XT].

The contents of byte 0 of doubleword element i of VSR[XB] are placed into byte 7 of doubleword element i of VSR[XT].

VSR Data Layout for xxbrd

| src | VSR[XB].dword[0] | VSR[XB].dword[1] |
| tgt | VSR[XT].dword[0] | VSR[XT].dword[1] |

Special Registers Altered:
None
VSX Vector Byte-Reverse Halfword XX2-form

\texttt{xxbrh \ XT,\XB}

\begin{tabular}{cccccc}
  & 60 & T & 7 & B & 475 & B[X] \\
 0 & 6 & 71 & 16 & 21 & \\
\end{tabular}

If 
\texttt{MSR.VSX}=0 then 
\texttt{VSX.Unavailable()}

\texttt{do \ i = 0 to 7}
  \texttt{vsrc \leftarrow VSR[32\times B+8].hword[i]}
  \texttt{do \ j = 0 to 1}
    \texttt{VSR[32\times T+T].hword[i].byte[j] \leftarrow vsrc.byte[1-j]}
  \texttt{end}
\texttt{end}

Let \texttt{XT} be the value \(32\times T + T\).
Let \texttt{XB} be the value \(32\times B + B\).

For each integer value \(i\) from 0 to 7, do the following.
The contents of byte 1 of halfword element \(i\) of \(VSR[XB]\) are placed into byte 0 of halfword element \(i\) of \(VSR[XT]\).

The contents of byte 0 of halfword element \(i\) of \(VSR[XB]\) are placed into byte 1 of halfword element \(i\) of \(VSR[XT]\).

\textbf{Special Registers Altered:}

None

\textbf{VSR Data Layout for xxbrh}

\begin{tabular}{cccccccccccc}
  \texttt{src \ VSR[\ XB].hword[0]} & \texttt{VSR[\ XB].hword[1]} & \texttt{VSR[\ XB].hword[2]} & \texttt{VSR[\ XB].hword[3]} & \texttt{VSR[\ XB].hword[4]} & \texttt{VSR[\ XB].hword[5]} & \texttt{VSR[\ XB].hword[6]} & \texttt{VSR[\ XB].hword[7]} \\
  \texttt{tgt \ VSR[\ XT].hword[0]} & \texttt{VSR[\ XT].hword[1]} & \texttt{VSR[\ XT].hword[2]} & \texttt{VSR[\ XT].hword[3]} & \texttt{VSR[\ XT].hword[4]} & \texttt{VSR[\ XT].hword[5]} & \texttt{VSR[\ XT].hword[6]} & \texttt{VSR[\ XT].hword[7]} \\
  0 & 16 & 32 & 48 & 64 & 80 & 96 & 112 & 127
\end{tabular}
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 15
end

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 15, do the following.
   The contents of byte sub-element 15-i of VSR[XB]
   are placed into byte sub-element i of VSR[XT].

Special Registers Altered:
   None
### VSX Vector Byte-Reverse Word XX2-form

**xxbrw**  
**XT, XB**

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>15</th>
<th>B</th>
<th>475</th>
<th>BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>20</td>
</tr>
</tbody>
</table>

- if MSR.VSX=0 then VSX_Unavailable()
- do i = 0 to 3
  - vsrc ← VSR[32xBX+B].word[i]
  - do j = 0 to 3
    - VSR[32xTX+T].word[j].byte[j] ← vsrc.byte[3-j]
  - end
- end

Let XT be the value 32xTX + T.
Let XB be the value 32xBX + B.

For each integer value i from 0 to 3, do the following.
- The contents of byte 3 of word element i of VSR[XB] are placed into byte 0 of word element i of VSR[XT].
- The contents of byte 2 of word element i of VSR[XB] are placed into byte 1 of word element i of VSR[XT].
- The contents of byte 1 of word element i of VSR[XB] are placed into byte 2 of word element i of VSR[XT].
- The contents of byte 0 of word element i of VSR[XB] are placed into byte 3 of word element i of VSR[XT].

**Special Registers Altered:**

None

### VSX Vector Evaluate 8RR:XX4-form

**xxeval**  
**XT,XA,XB,XC,IMM**

Premier:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>//</th>
<th>///</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>8</td>
<td>12</td>
<td>14</td>
<td>24</td>
</tr>
</tbody>
</table>

Suffix:

<table>
<thead>
<tr>
<th>34</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>1</th>
<th>0</th>
<th>//</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
<td>28</td>
<td>29</td>
</tr>
</tbody>
</table>

- if MSR.VSX=0 then VSX_Unavailable()
- src1 ← VSR[32xA+A]
- src2 ← VSR[32xB+B]
- src3 ← VSR[32xC+C]
- result ← (~src1&~src2&~src3 & qword_bit_splat(IMM.bit[0]) | ~src1&~src2& src3 & qword_bit_splat(IMM.bit[1]) | ~src1 src2&~src3 & qword_bit_splat(IMM.bit[2]) | ~src1 src2& src3 & qword_bit_splat(IMM.bit[3]) | src1&~src2&~src3 & qword_bit_splat(IMM.bit[4]) | src1&~src2& src3 & qword_bit_splat(IMM.bit[5]) | src1 src2&~src3 & qword_bit_splat(IMM.bit[6]) | src1 src2& src3 & qword_bit_splat(IMM.bit[7]))
- VSR[32xTX+T] ← result

For each integer value i, 0 to 127, do the following.
- Let j be the value of the concatenation of the contents of bit i of VSR[XA], bit i of VSR[XB], bit i of VSR[XC].
- The value of bit j of IMM is placed into bit i of VSR[XT].

See Table 145, “xxeval(A, B, C, IMM) Equivalent Functions,” on page 968 for the equivalent function evaluated by this instruction for any given value of IMM.

**Special Registers Altered:**

None

### VSR Data Layout for xxbrw

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
<td>127</td>
</tr>
</tbody>
</table>

### VSR Data Layout for xxeval

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB]</td>
</tr>
<tr>
<td>src3</td>
<td>VSR[XC]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT]</td>
</tr>
<tr>
<td>0</td>
<td>127</td>
</tr>
<tr>
<td>IMM</td>
<td>Ob...</td>
</tr>
<tr>
<td>------------</td>
<td>-------</td>
</tr>
<tr>
<td>0000000...</td>
<td>false</td>
</tr>
<tr>
<td>0000001...</td>
<td>and(B, nor(B, C))</td>
</tr>
<tr>
<td>0000010...</td>
<td>nor(A, nor(B, C))</td>
</tr>
<tr>
<td>0000011...</td>
<td>and(B, C)</td>
</tr>
<tr>
<td>0000100...</td>
<td>and(B, nor(A, C))</td>
</tr>
<tr>
<td>0000101...</td>
<td>nor(B, nor(A, C))</td>
</tr>
<tr>
<td>0000110...</td>
<td>nor(B, nor(A, C))</td>
</tr>
<tr>
<td>0000111...</td>
<td>nor(A, nor(B, C))</td>
</tr>
<tr>
<td>0001000...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0001001...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0001010...</td>
<td>nor(A, nor(B, C))</td>
</tr>
<tr>
<td>0001011...</td>
<td>nor(A, nor(B, C))</td>
</tr>
<tr>
<td>0001100...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0001101...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0001110...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0001111...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0010000...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0010001...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0010010...</td>
<td>nor(A, nor(B, C))</td>
</tr>
<tr>
<td>0010011...</td>
<td>nor(A, nor(B, C))</td>
</tr>
<tr>
<td>0010100...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0010101...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0010110...</td>
<td>nor(A, nor(B, C))</td>
</tr>
<tr>
<td>0010111...</td>
<td>nor(A, nor(B, C))</td>
</tr>
<tr>
<td>0011000...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0011001...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0011010...</td>
<td>nor(A, nor(B, C))</td>
</tr>
<tr>
<td>0011011...</td>
<td>nor(A, nor(B, C))</td>
</tr>
<tr>
<td>0011100...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0011101...</td>
<td>and(A, nor(B, C))</td>
</tr>
<tr>
<td>0011110...</td>
<td>nor(A, nor(B, C))</td>
</tr>
<tr>
<td>0011111...</td>
<td>nor(A, nor(B, C))</td>
</tr>
</tbody>
</table>
VSX Vector Extract Unsigned Word XX2-form

**xxextractuw XT,XB,UIM**

- if MSR.VSX=0 then VSX_Unavailable()
- \( \text{src} \leftarrow \text{VSR}[32 \times \text{BX} + \text{B}] . \text{byte}[\text{UIM}:\text{UIM}+3] \)
- \( \text{VSR}[32 \times \text{TX} + \text{T}] . \text{dword}[0] \leftarrow \text{EXTZ64} (\text{src}) \)
- \( \text{VSR}[32 \times \text{TX} + \text{T}] . \text{dword}[1] \leftarrow 0x0000_0000_0000_0000 \)

Let \( \text{XT} \) be the value \( 32 \times \text{TX} + \text{T} \).
Let \( \text{XB} \) be the value \( 32 \times \text{BX} + \text{B} \).

The contents of byte elements \( \text{UI M} : \text{UI M}+3 \) of \( \text{VSR}[\text{XB}] \) are placed into word element 1 of \( \text{VSR}[\text{XT}] \). The contents of the remaining word elements of \( \text{VSR}[\text{XT}] \) are set to 0.

If the value of \( \text{UI M} \) is greater than 12, the results are undefined.

**Special Registers Altered:**
- None

**VSR Data Layout for xxextractuw**

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
<th>VSR[XT].dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

VSX Vector Insert Word XX2-form

**xxinsertw XT,XB,UIM**

- if MSR.VSX=0 then VSX_Unavailable()
- \( \text{VSR}[32 \times \text{TX} + \text{T}] . \text{byte}[\text{UIM}:\text{UIM}+3] \leftarrow \text{VSR}[32 \times \text{BX} + \text{B}] . \text{bit}[32:63] \)

Let \( \text{XT} \) be the value \( 32 \times \text{TX} + \text{T} \).
Let \( \text{XB} \) be the value \( 32 \times \text{BX} + \text{B} \).

The contents of word element 1 of \( \text{VSR}[\text{XB}] \) are placed into byte elements \( \text{UI M} : \text{UI M}+3 \) of \( \text{VSR}[\text{XT}] \). The contents of the remaining byte elements of \( \text{VSR}[\text{XT}] \) are not modified.

If the value of \( \text{UI M} \) is greater than 12, the results are undefined.

**Special Registers Altered:**
- None

**VSR Data Layout for xxinsertw**

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
<th>VSR[XB].word[1]</th>
<th>unused</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>unused</td>
<td>VSR[XB].word[1]</td>
<td>unused</td>
<td>unused</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
</tr>
</tbody>
</table>

Chapter 7. Vector-Scalar Extension Facility 969
VSX Vector Generate PCV from Byte Mask
X-form

Let $XT$ be the value $32 \times TX + T$.

If $IMM=0b00000$, let $pcv$ be the permute control vector required to enable a left-indexed permute ($vperm$ or $xxperm$) to implement an expansion of the leftmost byte elements of a source vector into the byte elements of a result vector specified by the byte-element mask in $VSR[VRB+32]$.

If $IMM=0b00001$, let $pcv$ be the permute control vector required to enable a left-indexed permute ($vperm$ or $xxperm$) to implement a compression of the sparse byte elements in a source vector specified by the byte-element mask in $VSR[VRB+32]$ into the leftmost byte elements of a result vector.

If $IMM=0b00010$, let $pcv$ be the permute control vector required to enable a right-indexed permute ($vpermr$ or $xxpermr$) to implement an expansion of the rightmost byte elements of a source vector into the byte elements of a result vector specified by the byte-element mask in $VSR[VRB+32]$.

If $IMM=0b00011$, let $pcv$ be the permute control vector required to enable a right-indexed permute ($vpermr$ or $xxpermr$) to implement a compression of the sparse byte elements in a source vector specified by the byte-element mask in $VSR[VRB+32]$ into the rightmost byte elements of a result vector.

$pcv$ is placed into $VSR[XT]$.

Unused values of $IMM$ are reserved.

Special Registers Altered:
None
Programming Note

The following is an example of how a Load VSX Vector and Expand Byte, when using Big-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask Byte.

```
xgenpcvbm vPCV, vMASK, 0b00000    // generates the required permute control vector for Big-Endian expansion
vcntmbb vN, vMASK, 0b1           // calculates N, number of true byte-mask elements
lxvl vLD, EA, rN                  // loads N bytes
// Option 1: expand & merge
xxperm vT, vLD, vT, vPCV          // perform the expansion,
// specifying vT as 2nd source operand causes expanded load data to be
// merged into VSR[vT]
// Option 2: expand & zero
xxperm vT, vLD, vZERO, vPCV       // perform the expansion,
// specifying vZERO (vector of 0s) as 2nd source operand causes expanded
// load data to be placed into VSR[vT] with other elements set to 0
```

The following is an example of how a Load VSX Vector Expand Byte, when using Little-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask Byte.

```
xgenpcvbm vPCV, vMASK, 0b00010    // generates the required permute control vector for Big-Endian expansion
vcntmbb vN, vMASK, 0b1           // calculates N, number of true byte-mask elements
lxvl vLD, EA, rN                  // loads N bytes
// Option 1: expand & merge
xxperm vT, vLD, vT, vPCV          // perform the expansion,
// specifying vT as 2nd source operand causes expanded load data to be
// merged into VSR[vT]
// Option 2: expand & zero
xxperm vT, vLD, vZERO, vPCV       // perform the expansion,
// specifying vZERO (vector of 0s) as 2nd source operand causes expanded
// load data to be placed into VSR[vT] with other elements set to 0
```

The following is an example of how a VSX Vector Compress Byte and Store, when using Big-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask Byte.

```
xgenpcvbm vPCV, vMASK, 0b00001    // generates the required permute control vector for Big-Endian compression
vcntmbb vN, vMASK, 0b1           // calculates N, number of true byte-mask elements
xxperm vSD, vS, vS, vPCV          // perform the compression
stxvl vSD, rEA, rN                // store N bytes
```

The following is an example of how a VSX Vector Byte Compress and Store, when using Little-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask.

```
xgenpcvbm vPCV, vMASK, 0b00011    // generates the required permute control vector for Big-Endian compression
vcntmbb vN, vMASK, 0b1           // calculates N, number of true byte-mask elements
xxperm vSD, vS, vS, vPCV          // perform the compression
stxvl vSD, rEA, rN                // store N bytes
```
**VSX Vector Generate PCV from Halfword Mask X-form**

\[\text{xxgencvhm} \quad \text{XT}, \text{VRB}, \text{IMM}\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>IMM</th>
<th>VRB</th>
<th>917</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If \(\text{MSR.VSX}=0\) then \text{VSX.Unavailable}().

If \(\text{IMM}=0b00000\) then do // Big-Endian expansion

\[j \leftarrow 0\]

\[\text{do } i = 0 \text{ to } 7 \]

\[\text{if } \text{VSR[VRB+32].hword[i].bit[0]} = 1 \text{ then do}\]

\[\text{VSR[XT].hword[i].byte[0]} \leftarrow 2j + 0x00\]

\[\text{VSR[XT].hword[i].byte[1]} \leftarrow 2j + 0x01\]

\[j \leftarrow j + 1\]

\[\text{end}\]

\[\text{else do}\]

\[\text{VSR[XT].hword[i].byte[0]} \leftarrow 2i + 0x10\]

\[\text{VSR[XT].hword[i].byte[1]} \leftarrow 2i + 0x11\]

\[\text{end}\]

\[\text{end}\]

else if \(\text{IMM}=0b00001\) then do // Big-Endian compression

\[j \leftarrow 0\]

\[\text{do } i = 0 \text{ to } 7 \]

\[\text{if } \text{VSR[VRB+32].hword[i].bit[0]} = 1 \text{ then do}\]

\[\text{VSR[XT].hword[i].byte[0]} \leftarrow 2i + 0x00\]

\[\text{VSR[XT].hword[i].byte[1]} \leftarrow 2i + 0x01\]

\[j \leftarrow j + 1\]

\[\text{end}\]

\[\text{end}\]

\[\text{do } i = j \text{ to } 7\]

\[\text{VSR[XT].hword[i]} \leftarrow 0xUUUU\]

\[\text{end}\]

else if \(\text{IMM}=0b00010\) then do // Little-Endian expansion

\[j \leftarrow 0\]

\[\text{do } i = 0 \text{ to } 7 \]

\[\text{if } \text{VSR[VRB+32].hword[7-i].bit[0]} = 1 \text{ then do}\]

\[\text{VSR[XT].hword[7-i].byte[0]} \leftarrow 2i + 0x00\]

\[\text{VSR[XT].hword[7-i].byte[1]} \leftarrow 2i + 0x01\]

\[j \leftarrow j + 1\]

\[\text{end}\]

\[\text{else do}\]

\[\text{VSR[XT].hword[7-i].byte[0]} \leftarrow 2i + 0x10\]

\[\text{VSR[XT].hword[7-i].byte[1]} \leftarrow 2i + 0x11\]

\[\text{end}\]

\[\text{end}\]

else if \(\text{IMM}=0b00011\) then do // Little-Endian compression

\[j \leftarrow 0\]

\[\text{do } i = 0 \text{ to } 7 \]

\[\text{if } \text{VSR[VRB+32].hword[7-i].bit[0]} = 1 \text{ then do}\]

\[\text{VSR[XT].hword[7-i].byte[0]} \leftarrow 2i + 0x00\]

\[\text{VSR[XT].hword[7-i].byte[1]} \leftarrow 2i + 0x01\]

\[j \leftarrow j + 1\]

\[\text{end}\]

\[\text{end}\]

\[\text{do } i = j \text{ to } 7\]

\[\text{VSR[XT].hword[7-i]} \leftarrow 0xUUUU\]

\[\text{end}\]

Let XT be the value \(32 \times TX + T\).

If \(\text{IMM}=0b00000\), let \text{pcv} be the permute control vector required to enable a left-indexed permute (\text{vperm} or \text{xxperm}) to implement an expansion of the leftmost halfword elements of a source vector into the halfword elements of a result vector specified by the halfword-element mask in \text{VSR[VRB+32]}. If \(\text{IMM}=0b00001\), let \text{pcv} be the permute control vector required to enable a left-indexed permute (\text{vperm} or \text{xxperm}) to implement a compression of the sparse halfword elements in a source vector specified by the halfword-element mask in \text{VSR[VRB+32]} into the leftmost halfword elements of a result vector.

If \(\text{IMM}=0b00010\), let \text{pcv} be the permute control vector required to enable a right-indexed permute (\text{vpermr} or \text{xxpermr}) to implement an expansion of the rightmost halfword elements of a source vector into the halfword elements of a result vector specified by the halfword-element mask in \text{VSR[VRB+32]}. If \(\text{IMM}=0b00011\), let \text{pcv} be the permute control vector required to enable a right-indexed permute (\text{vpermr} or \text{xxpermr}) to implement a compression of the sparse halfword elements in a source vector specified by the halfword-element mask in \text{VSR[VRB+32]} into the rightmost halfword elements of a result vector.

\text{pcv} is placed into \text{VSR[XT]}.

Unused values of \text{IMM} are reserved.

**Special Registers Altered:**

None
The following is an example of how a Load VSX Vector Expand Halfword, when using Big-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask Halfword.

```
xngenpcvhm vPCV, vMASK, 0b000000        // generates the required permute control vector for Big-Endian expansion
vcntmbh  rN, vMASK, 0b1               // calculates N, number of true halfword-mask elements, adjusted to # of bytes
lxvl     vLD, EA, rN                 // loads 2xN bytes

// Option 1: expand & merge
xxperm   vT, vLD, vT, vPCV           // perform the expansion, specifying vT as 2nd source operand causes expanded load data to be merged into VSR[vT]

// Option 2: expand & zero
xxperm   vT, vLD, vZERO, vPCV        // perform the expansion, specifying vZERO (vector of 0s) as 2nd source operand causes expanded load data to be placed into VSR[vT] with other elements set to 0
```

The following is an example of how a Load VSX Vector Expand Halfword, when using Little-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask Halfword.

```
xngenpcvhm vPCV, vMASK, 0b00010        // generates the required permute control vector for Little-Endian expansion
vcntmbh  rN, vMASK, 0b1               // calculates N, number of true halfword-mask elements, adjusted to # of bytes
lxvl     vLD, EA, rN                 // loads 2xN bytes

// Option 1: expand & merge
xxpermr  vT, vLD, vT, vPCV           // perform the expansion, specifying vT as 2nd source operand causes expanded load data to be merged into VSR[vT]

// Option 2: expand & zero
xxpermr  vT, vLD, vZERO, vPCV        // perform the expansion, specifying vZERO (vector of 0s) as 2nd source operand causes expanded load data to be placed into VSR[vT] with other elements set to 0
```

The following is an example of how a Store VSX Vector Compress Halfword, when using Big-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask Halfword.

```
xngenpcvhm vPCV, vMASK, 0b00001        // generates the required permute control vector for Big-Endian compression
vcntmbh  rN, vMASK, 0b1               // calculates N, number of true halfword-mask elements, adjusted to # of bytes
xxperm   vSD, vS, vS, vPCV           // perform the compression
stxvl    vSD, rEA, rN                // store 2xN bytes
```

The following is an example of how a Store VSX Vector Compress Halfword, when using Little-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask Halfword.

```
xngenpcvhm vPCV, vMASK, 0b00011        // generates the required permute control vector for Little-Endian compression
vcntmbh  rN, vMASK, 0b1               // calculates N, number of true halfword-mask elements, adjusted to # of bytes
xxpermr  vSD, vS, vS, vPCV           // perform the compression
stxvl    vSD, rEA, rN                // store 2xN bytes
```
**VSX Vector Generate PCV from Word Mask X-form**

\texttt{xxgenpcvwm} \ XT,VRB,IMM

<table>
<thead>
<tr>
<th>0</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>948</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>IMM</td>
<td>VRB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if \( \text{MSR.VSX}=0 \) then VSX Unavailable;

if \( \text{IMM}=0b00000 \) then do \ // Big-Endian expansion
\( j \leftarrow 0 \)
\( \text{do } i = 0 \text{ to } 3 \)
\( \text{if } \text{VSR}[\text{VRB+32}].\text{word}[i].\text{bit}[0]=1 \text{ then do} \)
\( \text{VSR}[\text{XT}].\text{word}[i].\text{byte}[0] \leftarrow 4x + 0x00 \)
\( \text{VSR}[\text{XT}].\text{word}[i].\text{byte}[1] \leftarrow 4x + 0x01 \)
\( \text{VSR}[\text{XT}].\text{word}[i].\text{byte}[2] \leftarrow 4x + 0x02 \)
\( \text{VSR}[\text{XT}].\text{word}[i].\text{byte}[3] \leftarrow 4x + 0x03 \)
\( j \leftarrow j + 1 \)
\( \text{end} \)
\( \text{else do} \)
\( \text{VSR}[\text{XT}].\text{word}[i].\text{byte}[0] \leftarrow 4x + 0x10 \)
\( \text{VSR}[\text{XT}].\text{word}[i].\text{byte}[1] \leftarrow 4x + 0x11 \)
\( \text{VSR}[\text{XT}].\text{word}[i].\text{byte}[2] \leftarrow 4x + 0x12 \)
\( \text{VSR}[\text{XT}].\text{word}[i].\text{byte}[3] \leftarrow 4x + 0x13 \)
\( \text{end} \)
\( \text{end} \)

else if \( \text{IMM}=0b00001 \) then do \ // Big-Endian compression
\( j \leftarrow 0 \)
\( \text{do } i = 0 \text{ to } 3 \)
\( \text{if } \text{VSR}[\text{VRB+32}].\text{word}[3-i].\text{bit}[0]=1 \text{ then do} \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[3] \leftarrow 4x + 0x00 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[2] \leftarrow 4x + 0x01 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[1] \leftarrow 4x + 0x02 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[0] \leftarrow 4x + 0x03 \)
\( j \leftarrow j + 1 \)
\( \text{end} \)
\( \text{else do} \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[3] \leftarrow 0xUUUU\_UUUU \)
\( \text{end} \)
\( \text{end} \)

else if \( \text{IMM}=0b00010 \) then do \ // Little-Endian expansion
\( j \leftarrow 0 \)
\( \text{do } i = 0 \text{ to } 3 \)
\( \text{if } \text{VSR}[\text{VRB+32}].\text{word}[3-i].\text{bit}[0]=1 \text{ then do} \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[3] \leftarrow 4x + 0x00 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[2] \leftarrow 4x + 0x01 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[1] \leftarrow 4x + 0x02 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[0] \leftarrow 4x + 0x03 \)
\( j \leftarrow j + 1 \)
\( \text{end} \)
\( \text{else do} \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[3] \leftarrow 4x + 0x10 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[2] \leftarrow 4x + 0x11 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[1] \leftarrow 4x + 0x12 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[0] \leftarrow 4x + 0x13 \)
\( \text{end} \)
\( \text{end} \)

else if \( \text{IMM}=0b00011 \) then do \ // Little-Endian compression
\( j \leftarrow 0 \)
\( \text{do } i = 0 \text{ to } 3 \)
\( \text{if } \text{VSR}[\text{VRB+32}].\text{word}[3-i].\text{bit}[0]=1 \text{ then do} \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[3] \leftarrow 4x + 0x00 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[2] \leftarrow 4x + 0x01 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[1] \leftarrow 4x + 0x02 \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[0] \leftarrow 4x + 0x03 \)
\( j \leftarrow j + 1 \)
\( \text{end} \)
\( \text{else do} \)
\( \text{VSR}[\text{XT}].\text{word}[3-i].\text{byte}[3] \leftarrow 0xUUUU\_UUUU \)
\( \text{end} \)
\( \text{end} \)

Let \( XT \) be the value \( 32 \times TX + T \).

If \( \text{IMM}=0b00000 \), let \( pcv \) be the permute control vector required to enable a left-indexed permute (\textit{vperm} or \textit{xxperm}) to implement an expansion of the leftmost word elements of a source vector into the word elements of a result vector specified by the word-element mask in \( \text{VSR}[\text{VRB+32}] \).

If \( \text{IMM}=0b00001 \), let \( pcv \) be the permute control vector required to enable a left-indexed permute (\textit{vperm} or \textit{xxperm}) to implement a compression of the sparse word elements in a source vector specified by the word-element mask in \( \text{VSR}[\text{VRB+32}] \) into the leftmost word elements of a result vector.

If \( \text{IMM}=0b00010 \), let \( pcv \) be the permute control vector required to enable a right-indexed permute (\textit{vpermr} or \textit{xxpermr}) to implement an expansion of the rightmost word elements of a source vector into the word elements of a result vector specified by the word-element mask in \( \text{VSR}[\text{VRB+32}] \).

If \( \text{IMM}=0b00011 \), let \( pcv \) be the permute control vector required to enable a right-indexed permute (\textit{vpermr} or \textit{xxpermr}) to implement a compression of the sparse word elements in a source vector specified by the word-element mask in \( \text{VSR}[\text{VRB+32}] \) into the rightmost word elements of a result vector.

\( pcv \) is placed into \( \text{VSR}[\text{XT}] \).

Unused values of \( \text{IMM} \) are reserved.

**Special Registers Altered:** None
Chapter 7. Vector-Scalar Extension Facility

The following is an example of how a Load VSX Vector Expand Word, when using Big-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask Word.

```assembly
xxgenpcvwm vPCV, vMASK, 0b000000 // generates the required permute control vector for Big-Endian expansion
cntmbw rN, vMASK, 0b01 // calculates N, number of true word-mask elements, adjusted to # of bytes
lxvl vLD, EA, rN // loads 4xN bytes

// Option 1: expand & merge
xxperm vT, vLD, vT, vPCV // perform the expansion,
// specifying vT as 2nd source operand causes expanded load data to be
// merged into VSR[vT]

// Option 2: expand & zero
xxperm vT, vLD, vZERO, vPCV // perform the expansion,
// specifying vZERO (vector of 0s) as 2nd source operand causes expanded
// load data to be placed into VSR[vT] with other elements set to 0
```

The following is an example of how a Load VSX Vector Expand Word, when using Little-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask Word.

```assembly
xxgenpcvwm vPCV, vMASK, 0b00010 // generates the required permute control vector for Little-Endian expansion
cntmbw rN, vMASK, 0b01 // calculates N, number of true word-mask elements, adjusted to # of bytes
lxvl vLD, EA, rN // loads 4xN bytes

// Option 1: expand & merge
xxperm vT, vLD, vT, vPCV // perform the expansion,
// specifying vT as 2nd source operand causes expanded load data to be
// merged into VSR[vT]

// Option 2: expand & zero
xxperm vT, vLD, vZERO, vPCV // perform the expansion,
// specifying vZERO (vector of 0s) as 2nd source operand causes expanded
// load data to be placed into VSR[vT] with other elements set to 0
```

The following is an example of how a Store VSX Vector Compress Word, when using Big-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask Word.

```assembly
xxgenpcvwm vPCV, vMASK, 0b00001 // generates the required permute control vector for Big-Endian compression
cntmbw rN, vMASK, 0b01 // calculates N, number of true word-mask elements, adjusted to # of bytes
xxperm vSD, vS, vS, vPCV // perform the compression
stxvl vSD, rEA, rN // store 4xN bytes
```

The following is an example of how a Store VSX Vector Compress Word, when using Little-Endian byte-ordering, can be emulated using VSX Vector Generate PCV from Mask Word.

```assembly
xxgenpcvwm vPCV, vMASK, 0b00011 // generates the required permute control vector for Little-Endian compression
cntmbw rN, vMASK, 0b01 // calculates N, number of true word-mask elements, adjusted to # of bytes
xxperm vSD, vS, vS, vPCV // perform the compression
stxvl vSD, rEA, rN // store 4xN bytes
```
**VSX Vector Generate PCV from Doubleword Mask X-form**

`xxgenpcvdm XT,VRB,IMM`

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>T</th>
<th>IMM</th>
<th>VRB</th>
<th>949</th>
<th>XT</th>
</tr>
</thead>
</table>

- if MSR.VSX=0 then VSX_Unavailable() 
- if IMM=0b00000 then do // Big-Endian expansion
  - `j ← 0`
  - do `i = 0 to 1`
    - if VSR[VRB+32].dword[i].bit[0]=1 then do
      - VSR[XT].dword[i].byte[0] ← 8×j + 0x00
      - VSR[XT].dword[i].byte[1] ← 8×j + 0x01
      - VSR[XT].dword[i].byte[2] ← 8×j + 0x02
      - VSR[XT].dword[i].byte[3] ← 8×j + 0x03
      - VSR[XT].dword[i].byte[4] ← 8×j + 0x04
      - VSR[XT].dword[i].byte[5] ← 8×j + 0x05
      - VSR[XT].dword[i].byte[6] ← 8×j + 0x06
      - VSR[XT].dword[i].byte[7] ← 8×j + 0x07
      - `j ← j + 1`
    - else do
      - VSR[XT].dword[i].byte[0] ← 8×i + 0x10
      - VSR[XT].dword[i].byte[1] ← 8×i + 0x11
      - VSR[XT].dword[i].byte[2] ← 8×i + 0x12
      - VSR[XT].dword[i].byte[3] ← 8×i + 0x13
      - VSR[XT].dword[i].byte[4] ← 8×i + 0x14
      - VSR[XT].dword[i].byte[5] ← 8×i + 0x15
      - VSR[XT].dword[i].byte[6] ← 8×i + 0x16
      - VSR[XT].dword[i].byte[7] ← 8×i + 0x17
      - `j ← j + 1`
  - end
  - do `i = j to 1`
    - VSR[XT].dword[i] ← 0xUUUU_UUUU_UUUU_UUUU
  - end
- else if IMM=0b00001 then do // Little-Endian expansion
  - `j ← 0`
  - do `i = 0 to 1`
    - if VSR[VRB+32].dword[i].bit[0]=1 then do
      - VSR[XT].dword[i].byte[7] ← 8×j + 0x00
      - VSR[XT].dword[i].byte[6] ← 8×j + 0x01
      - VSR[XT].dword[i].byte[5] ← 8×j + 0x02
      - VSR[XT].dword[i].byte[4] ← 8×j + 0x03
      - VSR[XT].dword[i].byte[3] ← 8×j + 0x04
      - VSR[XT].dword[i].byte[2] ← 8×j + 0x05
      - VSR[XT].dword[i].byte[1] ← 8×j + 0x06
      - VSR[XT].dword[i].byte[0] ← 8×j + 0x07
      - `j ← j + 1`
    - else do
      - VSR[XT].dword[i].byte[7] ← 8×i + 0x10
      - VSR[XT].dword[i].byte[6] ← 8×i + 0x11
      - VSR[XT].dword[i].byte[5] ← 8×i + 0x12
      - VSR[XT].dword[i].byte[4] ← 8×i + 0x13
      - VSR[XT].dword[i].byte[3] ← 8×i + 0x14
      - VSR[XT].dword[i].byte[2] ← 8×i + 0x15
      - VSR[XT].dword[i].byte[1] ← 8×i + 0x16
      - VSR[XT].dword[i].byte[0] ← 8×i + 0x17
      - `j ← j + 1`
  - end
  - do `i = j to 1`
    - VSR[XT].dword[i] ← 0xUUUU_UUUU_UUUU_UUUU
  - end
- else if IMM=0b00010 then do // Big-Endian compression
  - `j ← 0`
  - do `i = 0 to 1`
    - if VSR[VRB+32].dword[1-1].bit[0]=1 then do
      - VSR[XT].dword[1-1].byte[0] ← 8×j + 0x00
      - VSR[XT].dword[1-1].byte[1] ← 8×j + 0x01
      - VSR[XT].dword[1-1].byte[2] ← 8×j + 0x02
      - VSR[XT].dword[1-1].byte[3] ← 8×j + 0x03
      - VSR[XT].dword[1-1].byte[4] ← 8×j + 0x04
      - VSR[XT].dword[1-1].byte[5] ← 8×j + 0x05
      - VSR[XT].dword[1-1].byte[6] ← 8×j + 0x06
      - VSR[XT].dword[1-1].byte[7] ← 8×j + 0x07
      - `j ← j + 1`
    - else do
      - VSR[XT].dword[1-1].byte[7] ← 8×i + 0x10
      - VSR[XT].dword[1-1].byte[6] ← 8×i + 0x11
      - VSR[XT].dword[1-1].byte[5] ← 8×i + 0x12
      - VSR[XT].dword[1-1].byte[4] ← 8×i + 0x13
      - VSR[XT].dword[1-1].byte[3] ← 8×i + 0x14
      - VSR[XT].dword[1-1].byte[2] ← 8×i + 0x15
      - VSR[XT].dword[1-1].byte[1] ← 8×i + 0x16
      - VSR[XT].dword[1-1].byte[0] ← 8×i + 0x17
      - `j ← j + 1`
  - end
  - do `i = j to 1`
    - VSR[XT].dword[i] ← 0xUUUU_UUUU_UUUU_UUUU
  - end
- else if IMM=0b00011 then do // Little-Endian compression
  - `j ← 0`
  - do `i = 0 to 1`
    - if VSR[VRB+32].dword[1-1].bit[0]=1 then do
      - VSR[XT].dword[1-1].byte[0] ← 8×j + 0x00
      - VSR[XT].dword[1-1].byte[1] ← 8×j + 0x01
      - VSR[XT].dword[1-1].byte[2] ← 8×j + 0x02
      - VSR[XT].dword[1-1].byte[3] ← 8×j + 0x03
      - VSR[XT].dword[1-1].byte[4] ← 8×j + 0x04
      - VSR[XT].dword[1-1].byte[5] ← 8×j + 0x05
      - VSR[XT].dword[1-1].byte[6] ← 8×j + 0x06
      - VSR[XT].dword[1-1].byte[7] ← 8×j + 0x07
      - `j ← j + 1`
    - else do
      - VSR[XT].dword[1-1].byte[7] ← 8×i + 0x10
      - VSR[XT].dword[1-1].byte[6] ← 8×i + 0x11
      - VSR[XT].dword[1-1].byte[5] ← 8×i + 0x12
      - VSR[XT].dword[1-1].byte[4] ← 8×i + 0x13
      - VSR[XT].dword[1-1].byte[3] ← 8×i + 0x14
      - VSR[XT].dword[1-1].byte[2] ← 8×i + 0x15
      - VSR[XT].dword[1-1].byte[1] ← 8×i + 0x16
      - VSR[XT].dword[1-1].byte[0] ← 8×i + 0x17
      - `j ← j + 1`
  - end
  - do `i = j to 1`
    - VSR[XT].dword[i] ← 0xUUUU_UUUU_UUUU_UUUU
  - end

Let XT be the value 32×TX + T.

If IMM=0b00000, let pcv be the permute control vector required to enable a left-indexed permute (vperm or xxperm) to implement an expansion of the leftmost doubleword elements of a source vector into the doubleword elements of a result vector specified by the doubleword-element mask in VSR[VRB+32].

If IMM=0b00001, let pcv be the the permute control vector required to enable a left-indexed permute (vperm or xxperm) to implement a compression of the sparse doubleword elements in a source vector specified by the doubleword-element mask in VSR[VRB+32] into the leftmost doubleword elements of a result vector.
If IMM=0b00010, let **pcv** be the permute control vector required to enable a right-indexed permute (**vpermr** or **xxpermr**) to implement an expansion of the rightmost doubleword elements of a source vector into the doubleword elements of a result vector specified by the doubleword-element mask in VSR[VRB+32].

If IMM=0b00011, let **pcv** be the permute control vector required to enable a right-indexed permute (**vpermr** or **xxpermr**) to implement a compression of the sparse doubleword elements in a source vector specified by the doubleword-element mask in VSR[VRB+32] into the rightmost doubleword elements of a result vector. **pcv** is placed into VSR[XT].

Special Registers Altered:
None

---

**Programming Note**

The following is an example of how a **Load VSX Vector Expand Doubleword**, when using Big-Endian byte-ordering, can be emulated using **VSX Vector Generate PCV from Mask Doubleword**.

```plaintext
xxgenpcvdm vPCV, vMASK, 0b00000   // generates the required permute control vector for Big-Endian expansion
vcntmbd rN, vMASK, 0b1          // calculates N, number of true doubleword-mask elements, adjusted to # of bytes
lxvl vLD, EA, rN                 // loads 8×N bytes

// Option 1: expand & merge
xxperm vT, vLD, vT, vPCV         // perform the expansion,
// specifying vT as 2nd source operand causes expanded load data to be
// merged into VSR[vT]

// Option 2: expand & zero
xxperm vT, vLD, vZERO, vPCV      // perform the expansion,
// specifying vZERO (vector of 0s) as 2nd source operand causes expanded
// load data to be placed into VSR[vT] with other elements set to 0
```

The following is an example of how a **Load VSX Vector Expand Doubleword**, when using Little-Endian byte-ordering, can be emulated using **VSX Vector Generate PCV from Mask Doubleword**.

```plaintext
xxgenpcvdm vPCV, vMASK, 0b00010   // generates the required permute control vector for Little-Endian expansion
vcntmbd rN, vMASK, 0b1          // calculates N, number of true doubleword-mask elements, adjusted to # of bytes
lxvl vLD, EA, rN                 // loads 8×N bytes

// Option 1: expand & merge
xxperm vT, vLD, vT, vPCV         // perform the expansion,
// specifying vT as 2nd source operand causes expanded load data to be
// merged into VSR[vT]

// Option 2: expand & zero
xxperm vT, vLD, vZERO, vPCV      // perform the expansion,
// specifying vZERO (vector of 0s) as 2nd source operand causes expanded
// load data to be placed into VSR[vT] with other elements set to 0
```

The following is an example of how a **Store VSX Vector Compress Doubleword**, when using Big-Endian byte-ordering, can be emulated using **VSX Vector Generate PCV from Mask Doubleword**.

```plaintext
xxgenpcvdm vPCV, vMASK, 0b00001   // generates the required permute control vector for Big-Endian compression
vcntmbd rN, vMASK, 0b1          // calculates N, number of true doubleword-mask elements, adjusted to # of bytes
xxperm vSD, vS, vS, vPCV        // perform the compression
stxvl vSD, rEA, rN              // store 8×N bytes
```

The following is an example of how a **Store VSX Vector Compress Doubleword**, when using Little-Endian byte-ordering, can be emulated using **VSX Vector Generate PCV from Mask Doubleword**.

```plaintext
xxgenpcvdm vPCV, vMASK, 0b00011   // generates the required permute control vector for Little-Endian compression
vcntmbd rN, vMASK, 0b1          // calculates N, number of true doubleword-mask elements, adjusted to # of bytes
xxperm vSD, vS, vS, vPCV        // perform the compression
stxvl vSD, rEA, rN              // store 8×N bytes
```
VSX Vector Logical AND XX3-form

xxland XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>130</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();

VSR[32×TX+T] ← VSR[32×AX+A] & VSR[32×BX+B]

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are ANDed with the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered

None

VSR Data Layout for xxland

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT]</td>
</tr>
</tbody>
</table>

VSR Data Layout for xxlandc

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT]</td>
</tr>
</tbody>
</table>

VSX Vector Logical AND with Complement XX3-form

xxlandc XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>130</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();

VSR[32×TX+T] ← VSR[32×AX+A] & ~VSR[32×BX+B]

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are ANDed with the complement of the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered

None
### VSX Vector Logical Equivalence XX3-form

#### xxleqv

**XT,XA,XB**

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>A</th>
<th>16</th>
<th>B</th>
<th>21</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

\[ VSR[32\times XT + T] \leftarrow VSR[32\times AX + A] \equiv VSR[32\times BX + B] \]

Let \( XT \) be the value \( 32 \times TX + T \).
Let \(XA\) be the value \( 32 \times AX + A \).
Let \(XB\) be the value \( 32 \times BX + B \).

The contents of \( VSR[XA] \) are exclusive-ORed with the contents of \( VSR[XB] \) and the complemented result is placed into \( VSR[XT] \).

Special Registers Altered:
None

#### VSR Data Layout for xxleqv

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT]</td>
</tr>
</tbody>
</table>

#### VSR Data Layout for xxlnand

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT]</td>
</tr>
</tbody>
</table>

### VSX Vector Logical NAND XX3-form

#### xxlnand

**XT,XA,XB**

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>A</th>
<th>16</th>
<th>B</th>
<th>178</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

\[ VSR[32\times XT + T] \leftarrow \neg ( VSR[32\times AX + A] \& VSR[32\times BX + B] ) \]

Let \( XT \) be the value \( 32 \times TX + T \).
Let \(XA\) be the value \( 32 \times AX + A \).
Let \(XB\) be the value \( 32 \times BX + B \).

The contents of \( VSR[XA] \) are ANDed with the contents of \( VSR[XB] \) and the complemented result is placed into \( VSR[XT] \).

Special Registers Altered:
None
**VSX Vector Logical OR with Complement XX3-form**

**xxlorc** XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>170</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();

VSR[32×TX + T] ← VSR[32×AX + A] | ¬VSR[32×BX + B]

Let XT be the value 32×TX + T.
LetXA be the value 32×AX + A.
LetXB be the value 32×BX + B.

The contents of VSR[XA] are ORed with the complement of the contents of VSR[XB] and the result is placed into VSR[XT].

**Special Registers Altered:**
None

**VSR Data Layout for xxlorc**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[XA]</td>
<td>VSR[XB]</td>
<td>VSR[XT]</td>
</tr>
</tbody>
</table>

**VSX Vector Logical NOR XX3-form**

**xxlnor** XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>162</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();

VSR[32×TX + T] ← ¬(VSR[32×AX + A] | VSR[32×BX + B])

Let XT be the value 32×TX + T.
LetXA be the value 32×AX + A.
LetXB be the value 32×BX + B.

The contents of VSR[XA] are ORed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

**Special Registers Altered:**
None

**VSR Data Layout for xxlnor**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[XA]</td>
<td>VSR[XB]</td>
<td>VSR[XT]</td>
</tr>
</tbody>
</table>
### VSX Vector Logical OR XX3-form

**xxlor**  
**XT,XA,XB**

<table>
<thead>
<tr>
<th>60</th>
<th>5</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>146</th>
<th>154</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>146</td>
<td>154</td>
</tr>
</tbody>
</table>

let XT be the value \(32 \times TX + T\).

let XA be the value \(32 \times AX + A\).

let XB be the value \(32 \times BX + B\).

the contents of VSR[XA] are ORed with the contents of VSR[XB] and the result is placed into VSR[XT].

**Special Registers Altered**

None

**VSR Data Layout for xxlor**

```
src1  VSR[XA]
src2  VSR[XB]
tgt   VSR[XT]
```

### VSX Vector Logical XOR XX3-form

**xxxor**  
**XT,XA,XB**

<table>
<thead>
<tr>
<th>60</th>
<th>5</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>154</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>154</td>
</tr>
</tbody>
</table>

let XT be the value \(32 \times TX + T\).

let XA be the value \(32 \times AX + A\).

let XB be the value \(32 \times BX + B\).

the contents of VSR[XA] are exclusive-ORed with the contents of VSR[XB] and the result is placed into VSR[XT].

**Special Registers Altered**

None

**VSR Data Layout for xxxor**

```
src1  VSR[XA]
src2  VSR[XB]
tgt   VSR[XT]
```
### VSX Vector Merge High Word XX3-form

**xxmrghw**  
**XT,XA,XB**

<table>
<thead>
<tr>
<th>60</th>
<th>56</th>
<th>48</th>
<th>32</th>
<th>24</th>
<th>16</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>A</td>
<td>B</td>
<td>XA</td>
<td>TX</td>
<td>AX</td>
<td>BX</td>
<td>T</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable();

VSR[32×TX+T].word[0] ← VSR[32×AX+A].word[0]
VSR[32×TX+T].word[1] ← VSR[32×BX+B].word[0]

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
LetXB be the value 32×BX + B.

The contents of word element 0 of VSR[XA] are placed into word element 0 of VSR[XT].
The contents of word element 0 of VSR[XB] are placed into word element 1 of VSR[XT].
The contents of word element 1 of VSR[XA] are placed into word element 2 of VSR[XT].
The contents of word element 1 of VSR[XB] are placed into word element 3 of VSR[XT].

**Special Registers Altered**

None

**VSR Data Layout for xxmrghw**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[XA].word[0]</th>
<th>VSR[XA].word[1]</th>
<th>unused</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[XB].word[0]</td>
<td>VSR[XB].word[1]</td>
<td>unused</td>
<td>unused</td>
</tr>
</tbody>
</table>

### VSX Vector Merge Low Word XX3-form

**xxmrglw**  
**XT,XA,XB**

<table>
<thead>
<tr>
<th>60</th>
<th>56</th>
<th>48</th>
<th>32</th>
<th>24</th>
<th>16</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>A</td>
<td>B</td>
<td>XA</td>
<td>TX</td>
<td>AX</td>
<td>BX</td>
<td>T</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable();

VSR[32×TX+T].word[0] ← VSR[32×AX+A].word[2]

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
LetXB be the value 32×BX + B.

The contents of word element 2 of VSR[XA] are placed into word element 0 of VSR[XT].
The contents of word element 2 of VSR[XB] are placed into word element 1 of VSR[XT].
The contents of word element 3 of VSR[XA] are placed into word element 2 of VSR[XT].
The contents of word element 3 of VSR[XB] are placed into word element 3 of VSR[XT].

**Special Registers Altered**

None

**VSR Data Layout for xxmrglw**

|------|--------|--------|----------------|----------------|
VSX Move From Accumulator X-form

```
xmfacc  AS
  |  31 | AS  //  0  //  177  //  31 |
  |  0  |  6  9  11  16  21  31  177  |
```

```
if MSR.VSX=0 then VSX_Unavailable();

VSR[4×AS] ← ACC[AS][0]
VSR[4×AS+1] ← ACC[AS][1]
VSR[4×AS+2] ← ACC[AS][2]
VSR[4×AS+3] ← ACC[AS][3]
```

For each integer value \( i \) from 0 to 3, the contents of row \( i \) of \( ACC[AS] \) are placed into \( VSR[4×AS+i] \).

**Special Registers Altered:**
None

---

**Programming Note**

During extended periods of the execution of an application when there isn't any active use of the Accumulators and VSX Vector GER instructions, hardware may deactivate these facilities for power savings. Once deactivated, while any attempted execution of any \( xmfacc, xxmtacc, xxsetaccz \), or VSX Vector GER instruction will cause these facilities to become reactivated, this reactivation causes significant delay beyond the normal execution of these instructions. This delay can be avoided by periodically issuing an \( xxmfacc \) with \( AS=0 \) instruction during extended times that the facilities are not being used to keep the facilities activated. Since the contents of \( ACC[0] \) will be undefined after the first execution, performance on subsequent executions of \( xxmfacc 0 \) can be expected to be degraded compared to performance when the contents of \( ACC[0] \) are defined. As such, to keep the facilities activated, \( xxmfacc 0 \) should be used with attention to performance implications.
**VSX Move To Accumulator X-form**

```
xxmtacc AT

if MSR.VSX=0 then VSX_Unavailable()

ACC[AT][0] ← VSR[4×AT]
ACC[AT][1] ← VSR[4×AT+1]
ACC[AT][2] ← VSR[4×AT+2]
ACC[AT][3] ← VSR[4×AT+3]
```

For each integer value \(i\) from 0 to 3, the contents of \(VSR[4×AT+i]\) are placed into row \(i\) of \(ACC[AT]\).

**Special Registers Altered:**

None

**VSR Data Layout for xxmtacc**

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[4×AT]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VSR[4×AT+1]</td>
</tr>
<tr>
<td></td>
<td>VSR[4×AT+2]</td>
</tr>
<tr>
<td></td>
<td>VSR[4×AT+3]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt</th>
<th>ACC[AT][0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACC[AT][1]</td>
</tr>
<tr>
<td></td>
<td>ACC[AT][2]</td>
</tr>
<tr>
<td></td>
<td>ACC[AT][3]</td>
</tr>
</tbody>
</table>
**VSX Vector Permute XX3-form**

```
xperm  XT,XA XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

src.byte[0:15] ← VSR[32×AX+A]
src.byte[16:31] ← VSR[32×TX+T]
pcv.byte[0:15] ← VSR[32×BX+B]

do i = 0 to 15
idx ← pcv.byte[i].bit[3:7]
VSR[32×TX+T].byte[i] ← src.byte[idx]
end

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XT be the value 32×TX + T.

Let bytes 0:15 of src be the contents of VSR[XA].
Let bytes 16:31 of src be the contents of VSR[XT].

Let the permute control vector pcv be the contents of VSR[XB].

For each integer value i from 0 to 15, do the following.
Let idx be the unsigned integer in bits 3:7 of byte element i of pcv.

The contents of byte element idx of src is placed into byte element i of VSR[XT].

**Special Registers Altered:**

None

**VSR Data Layout for xxperm**

|------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|

| 0   | 8   | 16  | 24  | 32  | 40  | 48  | 56  | 64  | 72  | 80  | 88  | 96  | 104 | 112 | 120 | 127 |

**VSX Vector Permute Right-indexed XX3-form**

```
xpermr XT,XA XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>58</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

src.byte[0:15] ← VSR[32×AX+A]
src.byte[16:31] ← VSR[32×TX+T]
pcv.byte[0:15] ← VSR[32×BX+B]

do i = 0 to 15
idx ← pcv.byte[i].bit[3:7]
VSR[32×TX+T].byte[i] ← src.byte[31-idx]
end

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XT be the value 32×TX + T.

Let bytes 0:15 of src be the contents of VSR[XA].
Let bytes 16:31 of src be the contents of VSR[XT].

Let the permute control vector pcv be the contents of VSR[XB].

For each integer value i from 0 to 15, do the following.
Let idx be the unsigned integer in bits 3:7 of byte element i of pcv.

The contents of byte element 31-idx of src is placed into byte element i of VSR[XT].

**Special Registers Altered:**

None

**VSR Data Layout for xxpermr**

|------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|

| 0   | 8   | 16  | 24  | 32  | 40  | 48  | 56  | 64  | 72  | 80  | 88  | 96  | 104 | 112 | 120 | 127 |
**VSX Vector Permute Doubleword Immediate**

**XX3-form**

```
xxpermdi XT,XA,XB,DM
```

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>0</th>
<th>DM</th>
<th>10</th>
<th>VB</th>
<th>IX</th>
</tr>
</thead>
</table>

* if MSR.VSX=0 then VSX_Unavailable();

* \( VSR[32\times T+T].dword[0] \leftarrow VSR[32\times A+A].dword[DM.bit[0]] \)

* \( VSR[32\times T+T].dword[1] \leftarrow VSR[32\times B+B].dword[DM.bit[1]] \)

Let \( XT \) be the value \( 32\times TX + T \).

Let \( XA \) be the value \( 32\times AX + A \).

Let \( XB \) be the value \( 32\times BX + B \).

If \( DM.bit[0]=0 \), the contents of doubleword element 0 of \( VSR[XA] \) are placed into doubleword element 0 of \( VSR[XT] \). Otherwise the contents of doubleword element 1 of \( VSR[XA] \) are placed into doubleword element 0 of \( VSR[XT] \).

If \( DM.bit[1]=0 \), the contents of doubleword element 0 of \( VSR[XB] \) are placed into doubleword element 1 of \( VSR[XT] \). Otherwise the contents of doubleword element 1 of \( VSR[XB] \) are placed into doubleword element 1 of \( VSR[XT] \).

**Special Registers Altered**

None

**Extended Mnemonics**

None

**Extended mnemonics for VSX Vector Permute Doubleword Immediate:**

- \( xxspltd \) T,A,0  \( xxpermdi \) T,A,0000
- \( xxspltd \) T,A,1  \( xxpermdi \) T,A,0001
- \( xxmrghd \) T,A,B  \( xxpermdi \) T,A,B,0000
- \( xxmrghd \) T,A,B  \( xxpermdi \) T,A,B,0001
- \( xxswapd \) T,A  \( xxpermdi \) T,A,0010

**VSR Data Layout for xxpermdi**

<table>
<thead>
<tr>
<th>src1</th>
<th>( VSR[XA].dword[0] )</th>
<th>( VSR[XA].dword[1] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>( VSR[XA].dword[0] )</td>
<td>( VSR[XA].dword[1] )</td>
</tr>
<tr>
<td>tgt</td>
<td>( VSR[XT].dword[0] )</td>
<td>( VSR[XT].dword[1] )</td>
</tr>
</tbody>
</table>
VSX Vector Permute Extended 8RR:XX4-form

xxpermX XT,XA,XB,XC,UIM

Prefix:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>//</th>
<th>//</th>
<th>//</th>
<th>//</th>
<th>UIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>8</td>
<td>12</td>
<td>14</td>
<td>29</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

Suffix:

| 34 | 6 | 11 | 16 | 21 | 28 | 29 | 30 | 31 |

if MSR.VSX=0 then VSX_Unavailable()

src.qword[0] ← VSR[32×AX+A]
src.qword[1] ← VSR[32×BX+B]

do i = 0 to 15
  section ← VSR[32×CX+C].byte[i].bit[0:2]
  if section=UIM then
    VSR[32×TX+T].byte[i] ← src.byte[eidx]
  else
    VSR[32×TX+T].byte[i] ← 0x00
  end

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XC be the value 32×CX + C.

Let UIM specify which 32-byte section of the long vector that src contains.

Let src be the concatenation VSR[XA] and VSR[XB], comprising a 32-byte section of up to a 128-byte vector.

For each integer value i from 0 to 15, do the following.

  Let eidx be the contents of bits 3:7 of byte element i of VSR[XC].
  jr

  if UIM is equal to the contents of bits 0:2 of byte element i of VSR[XC], the contents of byte element eidx of src are placed into byte element i of VSR[XT]. Otherwise, the contents of byte element i of VSR[XT] are set to 0.

Special Registers Altered:

None

VSR Data Layout for xxpermX

Programming Note

The following is an example of emulating 256-bit xxperm, where a 256-bit vector is contained in a pair of VSRs. The instruction is capable of emulating up to a 1024-bit xxperm.

vTA = xxperm(vA0, vA1, vC0, 0);
vTB = xxperm(vB0, vB1, vC0, 1);
vTC0 = xxperm(vA0, vA1, vC1, 0);
vTC1 = xxperm(vB0, vB1, vC1, 1);

vT0 = xxlor(vTA0, vTB);
vT1 = xxlor(vTC0, vTC1);

Programming Note

The following is an example of a parallel table lookup. In this case, a 16-way SIMD 256-entry byte table lookup.

vT0 = xxperm(vS0, vS1, vINDEX, 0);
vT1 = xxperm(vS2, vS3, vINDEX, 1);
vT2 = xxperm(vS4, vS5, vINDEX, 2);
vT3 = xxperm(vS6, vS7, vINDEX, 3);
vT4 = xxperm(vS8, vS9, vINDEX, 4);
vT5 = xxperm(vSA, vSB, vINDEX, 5);
vT6 = xxperm(vSC, vSD, vINDEX, 6);
vT7 = xxperm(vSE, vSF, vINDEX, 7);

vT0 = xxlor(vT0, vT1);
vT1 = xxlor(vT2, vT3);
vT2 = xxlor(vT4, vT5);
vT3 = xxlor(vT6, vT7);

vT0 = xxlor(vT0, vT1);
vT1 = xxlor(vT2, vT3);

Chapter 7. Vector-Scalar Extension Facility 987
VSX Vector Select XX4-form

xxsel   XT,XA,XB,XC

if MSR.VSX=0 then VSX_Unavailable();

src1 ← VSR[32×AX+A]
src2 ← VSR[32×BX+B]
mask ← VSR[32×CX+C]

VSR[32×TX+T] ← (src1 & ~mask) | (src2 & mask)

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XC be the value 32×CX + C.

Let src1 be the contents of VSR[XA].
Let src2 be the contents of VSR[XB].
Let mask be the contents of VSR[XC].

The value, (src1 & ~mask) | (src2 & mask), is placed into VSR[XT].

Special Registers Altered
None

VSR Data Layout for xxsel

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR(XA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR(XB)</td>
</tr>
<tr>
<td>src3</td>
<td>VSR(XC)</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR(XT)</td>
</tr>
</tbody>
</table>
VSX Set Accumulator to Zero X-form

`xxsetaccz` AT

```plaintext
if MSR.VSX=0 then VSX_Unavailable()  

ACC[AT][0] ← 0x0000_0000_0000_0000_0000_0000_0000_0000  
ACC[AT][1] ← 0x0000_0000_0000_0000_0000_0000_0000_0000  
ACC[AT][2] ← 0x0000_0000_0000_0000_0000_0000_0000_0000  
ACC[AT][3] ← 0x0000_0000_0000_0000_0000_0000_0000_0000
```

For each integer value `i` from 0 to 3, the contents of row `i` of ACC[AT] are set to 0.

**Special Registers Altered:**

None

VSR Data Layout for `xxmtacc`

```plaintext
<table>
<thead>
<tr>
<th>tgt</th>
<th>ACC[AT][0] = 0x0000_0000_0000_0000_0000_0000_0000_0000</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ACC[AT][1] = 0x0000_0000_0000_0000_0000_0000_0000_0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACC[AT][2] = 0x0000_0000_0000_0000_0000_0000_0000_0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACC[AT][3] = 0x0000_0000_0000_0000_0000_0000_0000_0000</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
VSX Vector Shift Left Double by Word
Immediate XX3-form

xxsldwi XT,XA,XB,SHW

if MSR.VSX=0 then VSX_Unavailable()

source.qword[0] ← VSR[32×AX+A]
source.qword[1] ← VSR[32×BX+B]
VSR[32×TX+T] ← source.word[SHW:SHW+3]

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let vsr be the concatenation of the contents of VSR[XA] followed by the contents of VSR[XB].

Words SHW:SHW+3 of vsr are placed into VSR[XT].

Special Registers Altered
None

VSR Data Layout for xxslidwi

|------|----------------|----------------|----------------|----------------|
VSX Vector Splat Immediate Byte X-form

`xxspltib XT,IMM8`

```
<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>O</th>
<th>IMM8</th>
<th>360</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>13</td>
<td>21</td>
</tr>
</tbody>
</table>
```

- If TX=0 & MSR.VSX=0 then VSX_Unavailable()
- If TX=1 & MSR.VEC=0 then Vector_Unavailable()

```latex
do i = 0 to 15
  VSR[32×TX+T].byte[i] ← UIM8
end
```

Let XT be the sum 32×TX + T.

The value IMM8 is copied into each byte element of VSR[XT].

**Special Registers Altered:**
None

VSX Vector Splat Immediate Double-Precision 8RR:D-form

`xxspltidp XT,IMM32`

```
| 0 | 6 | 8 | 12 | 14 | 16 | 31 |
```

Prefix:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>imm0</th>
</tr>
</thead>
</table>

SUFFIX:

<table>
<thead>
<tr>
<th>32</th>
<th>T</th>
<th>2</th>
<th>4</th>
<th>66</th>
<th>31</th>
</tr>
</thead>
</table>

- If MSR.VSX=0 then VSX_Unavailable()

| IMM32 ← imm0≪16 | imm1 |

| temp ← bfp_CONVERT_FROM_BFP32(IMM32); |

| VSR[32×TX+T].dword[0] ← bfp64_CONVERT_FROM_BFP(IMM32); |

| VSR[32×TX+T].dword[1] ← bfp64_CONVERT_FROM_BFP(IMM32); |

Let IMM32 be the concatenation of imm0 and imm1, representing a single-precision value.

IMM32 is converted to double-precision format and placed into each doubleword element of VSR[XT].

If IMM32 specifies a single-precision denormal value (i.e., bits 1:8 equal to 0 and bits 9:31 not equal to 0), the result is undefined.

**Special Registers Altered:**
None

VSR Data Layout for `xxspltib`

```
0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127
```

VSR Data Layout for `xxspltidp`

```
tgt VSR[XT].dword[0] 0x0000_0000_0000_0000
0 64
```
### VSX Vector Splat Immediate Word 8RR:D-form

**xxspltiw**  
**XT, IMM32**

**Prefix:**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>//</td>
<td>//</td>
</tr>
</tbody>
</table>

**Suffix:**

<table>
<thead>
<tr>
<th>32</th>
<th>T</th>
<th>3</th>
<th>imm1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>1516</td>
</tr>
</tbody>
</table>

```
if MSR.VSX=0 then VSX_Unavailable();
IMM32 ← imm0<<16 | imm1
```

```
do i = 0 to 3
    VSR[32×TX+T].word[i] ← IMM32
end
```

Let XT be the value $32 \times TX + T$.
Let IMM32 be the concatenation of imm0 and imm1.

IMM32 is placed into each word element of VSR[XT].

**Special Registers Altered:**

None

### VSR Data Layout for xxspltiw

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### VSX Vector Splat Immediate32 Doubleword Indexed 8RR:D-form

**xxsplti32dx**  
**XT, IX, IMM32**

**Prefix:**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>//</td>
<td>//</td>
</tr>
</tbody>
</table>

**Suffix:**

<table>
<thead>
<tr>
<th>32</th>
<th>T</th>
<th>0</th>
<th>imm1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>1516</td>
</tr>
</tbody>
</table>

```
if MSR.VSX=0 then VSX_Unavailable();
IMM32 ← imm0<<16 | imm1
```

```
VSR[32×TX+T].word[0].word[IX] ← IMM32
VSR[32×TX+T].word[1].word[IX] ← IMM32
```

Let XT be the value $32 \times TX + T$.
Let IMM32 be the concatenation of imm0 and imm1.

IMM32 is placed into word element IX of each doubleword element of VSR[XT]. The contents of the remaining word elements are not modified.

**Special Registers Altered:**

None

### VSR Data Layout for xxsplti32dx

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**VSX Vector Splat Word XX2-form**

`xxspltw` XT,XB,UIM

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>///</th>
<th>UIM</th>
<th>16</th>
<th>21</th>
<th>164</th>
<th>S17</th>
</tr>
</thead>
</table>

If MSR.VSX=0 then VSX_Unavailable().

VSR[32×TX+T].word[0] ← VSR[32×BX+B].word[UIM]
VSR[32×TX+T].word[1] ← VSR[32×BX+B].word[UIM]

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of word element UIM of VSR[XB] are replicated in each word element of VSR[XT].

**Special Registers Altered**

None

**VSR Data Layout for xxspltw**

|-----|----------------|----------------|----------------|----------------|
Appendix A. Suggested Floating-Point Models

A.1 Floating-Point Round to Single-Precision Model

The following describes algorithmically the operation of the Floating Round to Single-Precision instruction.

If (FRB)_{1:11} < 897 and (FRB)_{1:63} > 0 then
  Do
    If FPSCR\_UE = 0 then goto Disabled Exponent Underflow
    If FPSCR\_UE = 1 then goto Enabled Exponent Underflow
  End

If (FRB)_{1:11} > 1150 and (FRB)_{1:11} < 2047 then
  Do
    If FPSCR\_OE = 0 then goto Disabled Exponent Overflow
    If FPSCR\_OE = 1 then goto Enabled Exponent Overflow
  End

If (FRB)_{1:11} > 896 and (FRB)_{1:11} < 1151 then goto Normal Operand
If (FRB)_{1:63} = 0 then goto Zero Operand
If (FRB)_{1:11} = 2047 then
  Do
    If (FRB)_{12:63} = 0 then goto Infinity Operand
    If (FRB)_{12} = 1 then goto QNaN Operand
    If (FRB)_{12} = 0 and (FRB)_{13:63} > 0 then goto SNaN Operand
  End

Disabled Exponent Underflow:
  sign ← (FRB)_0
  If (FRB)_{1:11} = 0 then
    Do
      exp ← -1022
      frac_{0:52} ← 0b0 || (FRB)_{12:63}
    End
  If (FRB)_{1:11} > 0 then
    Do
      exp ← (FRB)_{1:11} - 1023
      frac_{0:52} ← 0b1 || (FRB)_{12:63}
    End

Denormalize operand:
  G || R || X ← 0b000
  Do while exp < -126
    exp ← exp + 1
    frac_{0:52} || G || R || X ← 0b0 || frac_{0:52} || G || (R || X)
  End

FPSCR\_UX ← (frac_{24:52} || G || R || X) > 0
Round Single(sign, exp, frac_{0:52}, G, R, X)
FPSCR\_XX ← FPSCR\_UX | FPSCR\_FI
If frac_{0:52} = 0 then
  Do
    FRT_0 ← sign
    FRT_{1:63} ← 0
  End
If sign = 0 then FPSCR_FPRF ← "+ zero"
If sign = 1 then FPSCR_FPRF ← "- zero"
End
If frac0:52 > 0 then
  Do
    If frac0 = 1 then
      Do
        If sign = 0 then FPSCR_FPRF ← "+ normal number"
        If sign = 1 then FPSCR_FPRF ← "- normal number"
      End
    If frac0 = 0 then
      Do
        If sign = 0 then FPSCR_FPRF ← "+ denormalized number"
        If sign = 1 then FPSCR_FPRF ← "- denormalized number"
      End
    Normalize operand:
    Do while frac0 = 0
      exp ← exp - 1
      frac0:52 ← frac1:52 || 0b0
    End
    FRT0 ← sign
    FRT1:11 ← exp + 1023
    FRT12:63 ← frac1:52
  End
End
Done

Enabled Exponent Underflow:
FPSCR_UX ← 1
sign ← (FRB)_0
If (FRB)_1:11 = 0 then
  Do
    exp ← -1022
    frac0:52 ← 0b0 || (FRB)_12:63
  End
If (FRB)_1:11 > 0 then
  Do
    exp ← (FRB)_1:11 - 1023
    frac0:52 ← 0b1 || (FRB)_12:63
  End
Normalize operand:
Do while frac0 = 0
  exp ← exp - 1
  frac0:52 ← frac1:52 || 0b0
End
Round Single(sign,exp,frac0:52,0,0,0)
FPSCR_XX ← FPSCR_RX || FPSCR_FI
exp ← exp + 192
FRT0 ← sign
FRT1:11 ← exp + 1023
FRT12:63 ← frac1:52
If sign = 0 then FPSCR_FPRF ← "+ normal number"
If sign = 1 then FPSCR_FPRF ← "- normal number"
Done

Disabled Exponent Overflow:
FPSCR_OX ← 1
If FPSCR_RN = 0b00 then /* Round to Nearest */
  Do
    If (FRB)_0 = 0 then FRT ← 0x7FF0_0000_0000_0000
    If (FRB)_0 = 1 then FRT ← 0xFFFF_0000_0000_0000
    If (FRB)_0 = 0 then FPSCR_FPRF ← "+ infinity"
    If (FRB)_0 = 1 then FPSCR_FPRF ← "- infinity"
  End
If FPSCR_RN = 0b01 then /* Round toward Zero */
  Do

If (FRB)_0 = 0 then FRT \leftarrow 0x47EF_FFFF_E000_0000  
If (FRB)_0 = 1 then FRT \leftarrow 0xC7EF_FFFF_E000_0000  
If (FRB)_0 = 0 then FPSCR_{FPFR} \leftarrow "+ normal number"  
If (FRB)_0 = 1 then FPSCR_{FPFR} \leftarrow "- normal number"

End

If FPSCR_{RN} = 0b10 then
  /* Round toward +Infinity */
  Do
    If (FRB)_0 = 0 then FRT \leftarrow 0x7FF0_0000_0000_0000  
    If (FRB)_0 = 1 then FRT \leftarrow 0xC7EF_FFFF_E000_0000  
    If (FRB)_0 = 0 then FPSCR_{FPFR} \leftarrow "+ infinity"  
    If (FRB)_0 = 1 then FPSCR_{FPFR} \leftarrow "- normal number"
  End

If FPSCR_{RN} = 0b11 then
  /* Round toward -Infinity */
  Do
    If (FRB)_0 = 0 then FRT \leftarrow 0x47EF_FFFF_E000_0000  
    If (FRB)_0 = 1 then FRT \leftarrow 0xFFF0_0000_0000_0000  
    If (FRB)_0 = 0 then FPSCR_{FPFR} \leftarrow "+ normal number"  
    If (FRB)_0 = 1 then FPSCR_{FPFR} \leftarrow "- infinity"
  End

FPSCR_{FR} \leftarrow undefined  
FPSCR_I \leftarrow 1  
FPSCR_{XX} \leftarrow 1  
Done

Enabled Exponent Overflow:

\begin{align*}
  \text{sign} & \leftarrow (FRB)_0 \\
  \text{exp} & \leftarrow (FRB)_{1:11} - 1023 \\
  \text{frac}_{0:52} & \leftarrow 0b1 || (FRB)_{12:63} \\
  \text{Round Single} & (\text{sign,exp,frac}_{0:52},0,0,0) \\
  FPSCR_{XX} & \leftarrow FPSCR_{XX} | FPSCR_I
\end{align*}

Enabled Overflow:

\begin{align*}
  FPSCR_{OX} & \leftarrow 1 \\
  \text{exp} & \leftarrow \text{exp} - 192 \\
  \text{FRT}_0 & \leftarrow \text{sign} \\
  \text{FRT}_{1:11} & \leftarrow \text{exp} + 1023 \\
  \text{FRT}_{12:63} & \leftarrow \text{frac}_{1:52} \\
  \text{If sign} & = 0 \text{ then FPSCR}_{FPFR} \leftarrow "+ normal number" \\
  \text{If sign} & = 1 \text{ then FPSCR}_{FPFR} \leftarrow "- normal number"
\end{align*}

Done

Zero Operand:

\begin{align*}
  \text{FRT} & \leftarrow (FRB) \\
  \text{If (FRB)}_0 & = 0 \text{ then FPSCR}_{FPFR} \leftarrow "+ zero" \\
  \text{If (FRB)}_0 & = 1 \text{ then FPSCR}_{FPFR} \leftarrow "- zero" \\
  \text{FPSCR}_{FR} & \leftarrow 0b00 \\
\end{align*}

Done

Infinity Operand:

\begin{align*}
  \text{FRT} & \leftarrow (FRB) \\
  \text{If (FRB)}_0 & = 0 \text{ then FPSCR}_{FPFR} \leftarrow "+ infinity" \\
  \text{If (FRB)}_0 & = 1 \text{ then FPSCR}_{FPFR} \leftarrow "- infinity" \\
  \text{FPSCR}_{FR} & \leftarrow 0b00 \\
\end{align*}

Done

QNaN Operand:

\begin{align*}
  \text{FRT} & \leftarrow (FRB)_{0:34} || 290 \\
  \text{FPSCR}_{FPFR} & \leftarrow "QNaN" \\
  \text{FPSCR}_{FR} & \leftarrow 0b00 \\
\end{align*}

Done
SNaN Operand:

FPSCR VXSNAN ← 1
If FPSCR VE = 0 then
  Do
    FRT0:11 ← (FRB)0:11
    FRT12 ← 1
    FRT13:63 ← (FRB)13:34 || 290
    FPSCR FPRF ← “QNaN”
  End
FPSCR FR F1 ← 0b00
Done

Normal Operand:

sign ← (FRB)0
exp ← (FRB)1:11 - 1023
frac0:52 ← 0b1 || (FRB)12:63
Round Single(sign,exp,frac0:52,0,0,0)
FPSCR VX ← FPSCR VX | FPSCR FI
If exp > 127 and FPSCR OE = 0 then go to Disabled Exponent Overflow
If exp > 127 and FPSCR OE = 1 then go to Enabled Overflow
FRT0 ← sign
FRT1:11 ← exp + 1023
FRT12:63 ← frac1:52
If sign = 0 then FPSCR FPRF ← “+ normal number”
If sign = 1 then FPSCR FPRF ← “- normal number”
Done

Round Single(sign,exp,frac0:52,G,R,X):

inc ← 0
lsb ← frac23
gbit ← frac24
rbit ← frac25
xbit ← (frac26:52)||G||R||X≠0
If FPSCR RN = 0b00 then /* Round to Nearest */
  Do /* comparisons ignore u bits */
    If sign || lsb || gbit || rbit || xbit = 0bu1uu then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0bu01u then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0b011u then inc ← 1
  End
If FPSCR RN = 0b10 then /* Round toward + Infinity */
  Do /* comparisons ignore u bits */
    If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
  End
If FPSCR RN = 0b11 then /* Round toward – Infinity */
  Do /* comparisons ignore u bits */
    If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
  End
frac0:23 ← frac0:23 + inc
If carry_out = 1 then
  Do
    frac0:23 ← 0b1 || frac0:22
    exp ← exp + 1
  End
frac24:52 ← 290
FPSCR FR ← inc
FPSCR FI ← gbit | rbit | xbit
Return
A.2 Floating-Point Convert to Integer Model

The following describes algorithmically the operation of the *Floating Convert To Integer* instructions.

```plaintext
if Floating Convert To Integer Word then do
  round_mode ← FPSCR
  tgt_precision ← “32-bit signed integer”
end

if Floating Convert To Integer Word Unsigned then do
  round_mode ← FPSCR
  tgt_precision ← “32-bit unsigned integer”
end

if Floating Convert To Integer Word with round toward Zero then do
  round_mode ← 0b01
  tgt_precision ← “32-bit signed integer”
end

if Floating Convert To Integer Word Unsigned with round toward Zero then do
  round_mode ← 0b01
  tgt_precision ← “32-bit unsigned integer”
end

if Floating Convert To Integer Doubleword then do
  round_mode ← FPSCR
  tgt_precision ← “64-bit signed integer”
end

if Floating Convert To Integer Doubleword Unsigned then do
  round_mode ← FPSCR
  tgt_precision ← “64-bit unsigned integer”
end

if Floating Convert To Integer Doubleword with round toward Zero then do
  round_mode ← 0b01
  tgt_precision ← “64-bit signed integer”
end

if Floating Convert To Integer Doubleword Unsigned with round toward Zero then do
  round_mode ← 0b01
  tgt_precision ← “64-bit unsigned integer”
end

sign ← (FRB)0
if (FRB)1:11 = 2047 and (FRB)12:63 = 0 then goto Infinity Operand
if (FRB)1:11 = 2047 and (FRB)12 = 0 then goto SNaN Operand
if (FRB)1:11 = 2047 and (FRB)12 = 1 then goto QNaN Operand
if (FRB)1:11 > 1086 then goto Large Operand
if (FRB)1:11 > 0 then exp ← (FRB)1:11 - 1023 /* exp - bias */
if (FRB)1:11 = 0 then exp ← -1022
if (FRB)1:11 > 0 then frac0:64 ← 0b01 || (FRB)12:63 || 110 /* normal */
if (FRB)1:11 = 0 then frac0:64 ← 0b00 || (FRB)12:63 || 110 /* denormal */
gbit || rbit || xbit ← 0b000
do i=1,63-exp /* do the loop 0 times if exp = 63 */
  frac0:64 || gbit || rbit || xbit ← 0b0 || frac0:64 || gbit || (rbit | xbit)
done
Round Integer( sign, frac0:64, gbit, rbit, xbit, round_mode )
if sign = 1 then frac0:64 ← ¬frac0:64 + 1 /* needed leading 0 for -2^{64}<(FRB)<-2^{63} */
```
if \( \text{tgt\_precision} = \text{"32-bit signed integer"} \) and \( \text{frac0:64} > 2^{31} - 1 \) then

goto Large Operand

if \( \text{tgt\_precision} = \text{"64-bit signed integer"} \) and \( \text{frac0:64} > 2^{63} - 1 \) then

goto Large Operand

if \( \text{tgt\_precision} = \text{"32-bit signed integer"} \) and \( \text{frac0:64} < -2^{31} \) then

goto Large Operand

if \( \text{tgt\_precision} = \text{"64-bit signed integer"} \) and \( \text{frac0:64} < -2^{63} \) then

goto Large Operand

if \( \text{tgt\_precision} = \text{"32-bit unsigned integer"} \) & \( \text{frac0:64} > 2^{32} - 1 \) then

goto Large Operand

if \( \text{tgt\_precision} = \text{"64-bit unsigned integer"} \) & \( \text{frac0:64} > 2^{64} - 1 \) then

goto Large Operand

if \( \text{tgt\_precision} = \text{"32-bit unsigned integer"} \) & \( \text{frac0:64} < 0 \) then

goto Large Operand

if \( \text{tgt\_precision} = \text{"64-bit unsigned integer"} \) & \( \text{frac0:64} < 0 \) then

goto Large Operand

\[
\text{FPSCR}_{XX} \leftarrow \text{FPSCR}_{XX} | \text{FPSCR}_{F1}
\]

if \( \text{tgt\_precision} = \text{"32-bit signed integer"} \) then
\( \text{FRT} \leftarrow 0x\text{UUUU} \text{ UUUU} || \text{frac33:64} \)

if \( \text{tgt\_precision} = \text{"32-bit unsigned integer"} \) then
\( \text{FRT} \leftarrow 0x\text{UUUU} \text{ UUUU} || \text{frac33:64} \)

if \( \text{tgt\_precision} = \text{"64-bit signed integer"} \) then
\( \text{FRT} \leftarrow \text{frac1:64} \)

if \( \text{tgt\_precision} = \text{"64-bit unsigned integer"} \) then
\( \text{FRT} \leftarrow \text{frac1:64} \)

\[
\text{Round Integer}(\text{sign, frac0:64, gbit, rbit, xbit, round\_mode}):
\]

\[
\text{inc} \leftarrow 0
\]

if \( \text{round\_mode} = 0b00 \) then do /* Round to Nearest */

if \( \text{sign} || \text{frac64} || \text{gbit} || \text{rbit} || \text{xbit} = 0b\text{U11UU} \) then \( \text{inc} \leftarrow 1 \)

if \( \text{sign} || \text{frac64} || \text{gbit} || \text{rbit} || \text{xbit} = 0b\text{UU01U} \) then \( \text{inc} \leftarrow 1 \)

end

if \( \text{round\_mode} = 0b10 \) then do /* Round toward +Infinity */

if \( \text{sign} || \text{frac64} || \text{gbit} || \text{rbit} || \text{xbit} = 0b\text{U01UU} \) then \( \text{inc} \leftarrow 1 \)

if \( \text{sign} || \text{frac64} || \text{gbit} || \text{rbit} || \text{xbit} = 0b\text{UUU1U} \) then \( \text{inc} \leftarrow 1 \)

end

if \( \text{round\_mode} = 0b11 \) then do /* Round toward -Infinity */

if \( \text{sign} || \text{frac64} || \text{gbit} || \text{rbit} || \text{xbit} = 0b\text{U11UU} \) then \( \text{inc} \leftarrow 1 \)

if \( \text{sign} || \text{frac64} || \text{gbit} || \text{rbit} || \text{xbit} = 0b\text{UUU1U} \) then \( \text{inc} \leftarrow 1 \)

end

\( \text{frac0:64} \leftarrow \text{frac0:64} + \text{inc} \)

\( \text{FPSCR}_{FR} \leftarrow \text{inc} \)

\( \text{FPSCR}_{FI} \leftarrow \text{gbit} \ | \ \text{rbit} \ | \ \text{xbit} \)

\text{return}

\[
\text{Infinity Operand}:
\]

\( \text{FPSCR}_{FR} \leftarrow 0b0 \)

\( \text{FPSCR}_{FI} \leftarrow 0b0 \)

\( \text{FPSCR}_{VXCVT} \leftarrow 0b1 \)

if \( \text{FPSCR}_{VE} = 0 \) then do

if \( \text{tgt\_precision} = \text{"32-bit signed integer"} \) then do

if \( \text{sign}=0 \) then \( \text{FRT} \leftarrow 0x\text{UUUU} \text{ UUUU}_7\text{FFF} \text{ FFFF} \)

if \( \text{sign}=1 \) then \( \text{FRT} \leftarrow 0x\text{UUUU} \text{ UUUU}_8\text{000}_0\text{000} \)

end

else if \( \text{tgt\_precision} = \text{"32-bit unsigned integer"} \) then do

if \( \text{sign}=0 \) then \( \text{FRT} \leftarrow 0x\text{UUUU} \text{ UUUU}_7\text{FFF} \text{ FFFF} \)

if \( \text{sign}=1 \) then \( \text{FRT} \leftarrow 0x\text{UUUU} \text{ UUUU}_8\text{000}_0\text{000} \)

end

else if \( \text{tgt\_precision} = \text{"64-bit signed integer"} \) then do

if \( \text{sign}=0 \) then \( \text{FRT} \leftarrow 0x7\text{FFF} \text{ FFFF} \text{ FFFF} \text{ FFFF} \)

if \( \text{sign}=1 \) then \( \text{FRT} \leftarrow 0x8\text{000}_0\text{000}_0\text{000}_0\text{000} \)

end
end

else if tgt_precision = "64-bit unsigned integer" then do
  if sign=0 then FRT ← 0xFFFF_FFFF_FFFF_FFFF
  if sign=1 then FRT ← 0x0000_0000_0000_0000
end
FPSCR_{PPRF} ← 0bUUUUU
end
done

SNaN Operand:
FPSCR_{FR} ← 0b0
FPSCR_{FI} ← 0b0
FPSCR_{XSNAN} ← 0b1
FPSCR_{XCVI} ← 0b1
if FPSCR_{VE} = 0 then do
  if tgt_precision = "32-bit signed integer" then FRT ← 0xUUUU_UUUU_8000_0000
  if tgt_precision = "64-bit signed integer" then FRT ← 0x8000_0000_0000_0000
  if tgt_precision = "32-bit unsigned integer" then FRT ← 0xUUUU_UUUU_0000_0000
  if tgt_precision = "64-bit unsigned integer" then FRT ← 0x0000_0000_0000_0000
FPSCR_{PPRF} ← 0bUUUUU
end
done

QNaN Operand:
FPSCR_{FR} ← 0b0
FPSCR_{FI} ← 0b0
FPSCR_{XCVI} ← 0b1
if FPSCR_{VE} = 0 then do
  if tgt_precision = "32-bit signed integer" then FRT ← 0xUUUU_UUUU_7FFF_FFFF
  if tgt_precision = "64-bit signed integer" then FRT ← 0x8000_0000_0000_0000
  if tgt_precision = "32-bit unsigned integer" then FRT ← 0xUUUU_UUUU_8000_0000
  if tgt_precision = "64-bit unsigned integer" then FRT ← 0x0000_0000_0000_0000
FPSCR_{PPRF} ← 0bUUUUU
end
done

Large Operand:
FPSCR_{FR} ← 0b0
FPSCR_{FI} ← 0b0
FPSCR_{XCVI} ← 0b1
if FPSCR_{VE} = 0 then do
  if tgt_precision = "32-bit signed integer" then do
    if sign = 0 then FRT ← 0xUUUU_UUUU_7FFF_FFFF
    if sign = 1 then FRT ← 0xUUUU_UUUU_8000_0000
  end
else if tgt_precision = "64-bit signed integer" then do
  if sign = 0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
  if sign = 1 then FRT ← 0x8000_0000_0000_0000
end
else if tgt_precision = "32-bit unsigned integer" then do
  if sign = 0 then FRT ← 0xUUUU_UUUU_FFFF_FFFF
  if sign = 1 then FRT ← 0x8000_0000_0000_0000
end
else if tgt_precision = "64-bit unsigned integer" then do
  if sign = 0 then FRT ← 0xFFFF_FFFF_FFFF_FFFF
  if sign = 1 then FRT ← 0x0000_0000_0000_0000
end
FPSCR_{PPRF} ← 0bUUUUU
end
done
A.3 Floating-Point Convert from Integer Model

The following describes algorithmically the operation of the Floating Convert From Integer instructions.

if Floating Convert From Integer Doubleword then do
  tgt_precision ← "double-precision"
  sign ← (FRB)₀
  exp ← 63
  frac₀:63 ← (FRB)
end

if Floating Convert From Integer Doubleword Single then do
  tgt_precision ← "single-precision"
  sign ← (FRB)₀
  exp ← 63
  frac₀:63 ← (FRB)
end

if Floating Convert From Integer Doubleword Unsigned then do
  tgt_precision ← "double-precision"
  sign ← 0
  exp ← 63
  frac₀:63 ← (FRB)
end

if Floating Convert From Integer Doubleword Unsigned Single then do
  tgt_precision ← "single-precision"
  sign ← 0
  exp ← 63
  frac₀:63 ← (FRB)
end

if frac₀:63 = 0 then go to Zero Operand
if sign = 1 then frac₀:63 ← ¬frac₀:63 + 1

/* do the loop 0 times if (FRB) = max negative 64-bit integer or */
/*                     if (FRB) = max unsigned 64-bit integer    */
do while frac₀ = 0
  frac₀:63 ← frac₁:63 || 0b0
  exp ← exp - 1
end

Round Float( sign, exp, frac₀:63, RN )
if sign = 0 then FPSCR_FPRF ← "normal number"
if sign = 1 then FPSCR_FPRF ← "normal number"
FRT₀ ← sign
FRT₁:11 ← exp + 1023 /* exp + bias */
FRT₁₂:63 ← frac₁:52
done

Zero Operand:
FPSCR_FR ← 0b00
FPSCR_FI ← 0b00
FPSCR_FPRF ← "+ zero"
FRT ← 0x0000_0000_0000_0000
done

Round Float( sign, exp, frac₀:63, round_mode ):
  inc ← 0

  if tgt_precision = "single-precision" then do
    lsb ← frac₂₃
    gbit ← frac₂₄
    rbit ← frac₂₅
    xbit ← frac₂₆:63 > 0
  end
  else do /* tgt_precision = "double-precision" */
lsb ← frac_52
gbit ← frac_53
rbit ← frac_54
xbit ← frac_55:63 > 0
end

if round_mode = 0b00 then do                /* Round to Nearest */
    if sign || lsb || gbit || rbit || xbit = 0bU11UU then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0bU011U then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0bU01U1 then inc ← 1
end

if round_mode = 0b10 then do                /* Round toward + Infinity */
    if sign || lsb || gbit || rbit || xbit = 0b0U1UU then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b0UU1U then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b0UUU1 then inc ← 1
end

if round_mode = 0b11 then do                /* Round toward - Infinity */
    if sign || lsb || gbit || rbit || xbit = 0b1U1UU then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b1UU1U then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b1UUU1 then inc ← 1
end

if tgt_precision = “single-precision” then
    frac_0:23 ← frac_0:23 + inc
else /* tgt_precision = “double-precision” */
    frac_0:52 ← frac_0:52 + inc

if carry_out = 1 then exp ← exp + 1

FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
FPSCRX = FPSCRX | FPSCRFI
return
A.4 Floating-Point Round to Integer Model

The following describes algorithmically the operation of the Floating Round To Integer instructions.

If (FRB)\(_{1:11} = 2047\) and (FRB)\(_{12:63} = 0\), then goto Infinity Operand
If (FRB)\(_{1:11} = 2047\) and (FRB)\(_{12} = 0\), then goto SNaN Operand
If (FRB)\(_{1:11} = 2047\) and (FRB)\(_{12} = 1\), then goto QNaN Operand
if (FRB)\(_{1:63} = 0\) then goto Zero Operand
If (FRB)\(_{1:11} < 1023\) then goto Small Operand /* exp < 0; \mid value \mid < 1*/
If (FRB)\(_{1:11} > 1074\) then goto Large Operand /* exp > 51; integral value */

\[
\text{sign} \leftarrow (FRB)_{0}
\]
\[
\text{exp} \leftarrow (FRB)_{1:11} - 1023 \quad \text{/* exp - bias */}
\]
\[
\text{frac}_{0:52} \leftarrow 0b1 || (FRB)_{12:63}
\]
\[
\text{gbit} || \text{rbit} || \text{xbit} \leftarrow 0b000
\]

\[
\text{Do } i = 1, 52 - \exp \text{ End}
\]
Round Integer (sign, frac\(_{0:52}\), gbit, rbit, xbit)

\[
\text{Do } i = 2, 52 - \exp \text{ End}
\]
If frac\(_{0} = 1\), then exp \leftarrow exp + 1
Else frac\(_{0:52} \leftarrow \text{frac}_{1:52} || 0b0
\]

\[
\text{FRT}_{0} \leftarrow \text{sign}
\]
\[
\text{FRT}_{1:11} \leftarrow \exp + 1023
\]
\[
\text{FRT}_{12:63} \leftarrow \text{frac}_{1:52}
\]

If (FRT)\(_{0} = 0\) then FPSCR\(_{FPRF} \leftarrow \pm \text{normal number}
Else FPSCR\(_{FPRF} \leftarrow \mp \text{normal number}
FPSCR\(_{FR Fi} \leftarrow 0b00
Done

Round Integer(sign, frac\(_{0:52}\), gbit, rbit, xbit):

inc \leftarrow 0
If inst = Floating Round to Integer Nearest then /* ties away from zero */

Do /* comparisons ignore u bits */
If sign || frac\(_{52} || gbit || rbit || xbit = 0buu1uu then inc \leftarrow 1
End
If inst = Floating Round to Integer Plus then

Do /* comparisons ignore u bits */
If sign || frac\(_{52} || gbit || rbit || xbit = 0b0u1uu then inc \leftarrow 1
If sign || frac\(_{52} || gbit || rbit || xbit = 0buuu1 then inc \leftarrow 1
If sign || frac\(_{52} || gbit || rbit || xbit = 0b0uuu1 then inc \leftarrow 1
End
If inst = Floating Round to Integer Minus then

Do /* comparisons ignore u bits */
If sign || frac\(_{52} || gbit || rbit || xbit = 0b1uu1u then inc \leftarrow 1
If sign || frac\(_{52} || gbit || rbit || xbit = 0b1uuuu then inc \leftarrow 1
If sign || frac\(_{52} || gbit || rbit || xbit = 0b1uuuu1 then inc \leftarrow 1
End
frac\(_{0:52} \leftarrow \text{frac}_{0:52} + \text{inc}
Return
Infinity Operand:
FRT ← (FRB)
If (FRB)0 = 0 then FPSCR_FPRF ← "+ infinity"
If (FRB)0 = 1 then FPSCR_FPRF ← "- infinity"
FPSCR_FRFI ← 0b00
Done

SNaN Operand:
FPSCR_VXSNAN ← 1
If FPSCR_VE = 0 then
   Do
      FRT ← (FRB)
      FRT12 ← 1
      FPSCR_FPRF ← "QNaN"
   End
   FPSCR_FRFI ← 0b00
   Done

QNaN Operand:
FRT ← (FRB)
FPSCR_FPRF ← "QNaN"
FPSCR_FRFI ← 0b00
Done

Zero Operand:
If (FRB)0 = 0 then
   Do
      FRT ← 0x0000_0000_0000_0000
      FPSCR_FPRF ← "+ zero"
   End
Else
   Do
      FRT ← 0x8000_0000_0000_0000
      FPSCR_FPRF ← "- zero"
   End
   FPSCR_FRFI ← 0b00
   Done

Small Operand:
If inst = Floating Round to Integer Nearest and
(FRB)1:11 < 1022 then goto Zero Operand
If inst = Floating Round to Integer Toward Zero
then goto Zero Operand
If inst = Floating Round to Integer Plus and (FRB)0
= 1 then goto Zero Operand
If inst = Floating Round to Integer Minus and
(FRB)0 = 0 then goto Zero Operand
If (FRB)0 = 0 then
   Do
      /* value = 1.0 */
      FRT ← 0x3FF0_0000_0000_0000
      FPSCR_FPRF ← "+ normal number"
   End
Else
   Do
      /* value = -1.0 */
      FRT ← 0xBFF0_0000_0000_0000
      FPSCR_FPRF ← "- normal number"
   End
   FPSCR_FRFI ← 0b00
   Done

Large Operand:
FRT ← (FRB)
Appendix B. Densely Packed Decimal

The trailing significand field of the decimal floating-point data format is encoded using Densely Packed Decimal (DPD). DPD encoding is a compression technique which supports the representation of decimal integers of arbitrary length. Translation operates on three Binary Coded Decimal (BCD) digits at a time compressing the 12 bits into 10 bits with an algorithm that can be applied or reversed using simple Boolean operations. In the following examples, a 3-digit BCD number is represented as (abcd)(efgh)(ijkm), a 10-bit DPD number is represented as (pqr)(stu)(v)(wxy), and the Boolean operations, & (AND), | (OR), and ¬ (NOT) are used.

B.1 BCD-to-DPD Translation

The translation from a 3-digit BCD number to a 10-bit DPD can be performed through the following Boolean operations.

\[
\begin{align*}
p &= (f \& a \& i \& \neg e) \mid (j \& a \& \neg i) \mid (b \& \neg a) \\
qu &= (g \& a \& i \& \neg e) \mid (k \& a \& \neg i) \mid (c \& \neg a) \\
q &= d \\
s &= (j \& \neg a \& e \& \neg i) \mid (f \& \neg i \& \neg e) \\
t &= (k \& \neg a \& e \& \neg i) \mid (g \& \neg i \& \neg e) \\
u &= (j \& \neg a \& e \& \neg i) \mid (f \& \neg i \& \neg e) \\
v &= (j \& \neg a \& e \& \neg i) \mid (f \& \neg i \& \neg e) \\
w &= (j \& \neg a \& e \& \neg i) \mid (f \& \neg i \& \neg e) \\
x &= (j \& \neg a \& e \& \neg i) \mid (f \& \neg i \& \neg e) \\
y &= (j \& \neg a \& e \& \neg i)
\end{align*}
\]

Alternatively, the following table can be used to perform the translation. The most significant bit of the three BCD digits (left column) is used to select a specific 10-bit encoding (right column) of the DPD.

<table>
<thead>
<tr>
<th>ae</th>
<th>pqr</th>
<th>stu</th>
<th>v</th>
<th>wxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>bcd</td>
<td>fgh</td>
<td>0</td>
<td>jkm</td>
</tr>
<tr>
<td>001</td>
<td>bcd</td>
<td>fgh</td>
<td>0</td>
<td>00m</td>
</tr>
<tr>
<td>010</td>
<td>bcd</td>
<td>jkh</td>
<td>1</td>
<td>01m</td>
</tr>
<tr>
<td>011</td>
<td>bcd</td>
<td>jkh</td>
<td>0</td>
<td>01m</td>
</tr>
<tr>
<td>100</td>
<td>jkd</td>
<td>fgh</td>
<td>1</td>
<td>10m</td>
</tr>
<tr>
<td>101</td>
<td>fgd</td>
<td>01h</td>
<td>1</td>
<td>11m</td>
</tr>
<tr>
<td>110</td>
<td>jkd</td>
<td>00h</td>
<td>1</td>
<td>11m</td>
</tr>
<tr>
<td>111</td>
<td>00d</td>
<td>11h</td>
<td>1</td>
<td>11m</td>
</tr>
</tbody>
</table>

The full translation of a 3-digit BCD number (000 - 999) to a 10-bit DPD is shown in Table 146 on page 1009, with the DPD entries shown in hexadecimal format. The BCD number is produced by replacing ‘_’ in the leftmost column with the corresponding digit along the top row. The table is split into two halves, with the right half being a continuation of the left half.

B.2 DPD-to-BCD Translation

The translation from a 10-bit DPD to a 3-digit BCD number can be performed through the following Boolean operations.

\[
\begin{align*}
a &= (\neg s \& v \& w) \mid (t \& v \& w \& s) \mid (v \& w \& \neg x) \\
b &= (p \& s \& x \& \neg t) \mid (p \& \neg w) \mid (p \& \neg v) \\
c &= (q \& s \& x \& \neg t) \mid (q \& \neg w) \mid (q \& \neg v) \\
d &= r \\
e &= (v \& \neg w \& x) \mid (s \& v \& w \& x) \\
f &= (p \& t \& v \& w \& x \& \neg s) \mid (s \& \neg x \& v) \mid (s \& \neg v) \\
g &= (q \& t \& w \& v \& x \& \neg s) \mid (t \& \neg x \& v) \mid (t \& \neg v) \\
h &= u \\
i &= (t \& v \& w \& x) \mid (s \& v \& w \& x) \\
j &= (p \& s \& \neg t \& w \& v) \mid (s \& v \& \neg w \& x) \\
k &= (q \& s \& \neg t \& v \& w) \mid (t \& v \& \neg w \& x) \\
m &= y
\end{align*}
\]

Alternatively, the following table can be used to perform the translation. A combination of five bits in the DPD encoding (leftmost column) are used to specify a translation to the 3-digit BCD encoding. Dashes (-) in the table are don’t cares, and can be either one or zero.
The full translation of the 10-bit DPD to a 3-digit BCD number is shown in Table 147 on page 1010. The 10-bit DPD index is produced by concatenating the 6-bit value shown in the left column with the 4-bit index along the top row, both represented in hexadecimal. The values in parentheses are non-preferred translations and are explained further in the following section.

### B.3 Preferred DPD encoding

Translating from a 3-digit BCD number (1000 numbers) to a 10-bit DPD encoding (1024 combinations) leaves 24 redundant translations. The 24 redundant combinations are evenly assigned to eight BCD numbers and are shown in the following table, with the non-preferred encoding in parentheses. The preferred encoding is produced by translating a 3-digit BCD number with the translation table or Boolean operations shown in Section B.1. The redundant DPD encodings are all valid and will be correctly translated to their respective BCD value through the mechanisms provided in Section B.2. For decimal floating-point operations all DPD encodings are recognized as source operands.
<table>
<thead>
<tr>
<th>00_</th>
<th>000</th>
<th>001</th>
<th>002</th>
<th>003</th>
<th>004</th>
<th>005</th>
<th>006</th>
<th>007</th>
<th>008</th>
<th>009</th>
</tr>
</thead>
<tbody>
<tr>
<td>01_</td>
<td>010</td>
<td>011</td>
<td>012</td>
<td>013</td>
<td>014</td>
<td>015</td>
<td>016</td>
<td>017</td>
<td>018</td>
<td>019</td>
</tr>
<tr>
<td>02_</td>
<td>020</td>
<td>021</td>
<td>022</td>
<td>023</td>
<td>024</td>
<td>025</td>
<td>026</td>
<td>027</td>
<td>028</td>
<td>029</td>
</tr>
<tr>
<td>03_</td>
<td>030</td>
<td>031</td>
<td>032</td>
<td>033</td>
<td>034</td>
<td>035</td>
<td>036</td>
<td>037</td>
<td>038</td>
<td>039</td>
</tr>
<tr>
<td>04_</td>
<td>040</td>
<td>041</td>
<td>042</td>
<td>043</td>
<td>044</td>
<td>045</td>
<td>046</td>
<td>047</td>
<td>048</td>
<td>049</td>
</tr>
<tr>
<td>05_</td>
<td>050</td>
<td>051</td>
<td>052</td>
<td>053</td>
<td>054</td>
<td>055</td>
<td>056</td>
<td>057</td>
<td>058</td>
<td>059</td>
</tr>
<tr>
<td>06_</td>
<td>060</td>
<td>061</td>
<td>062</td>
<td>063</td>
<td>064</td>
<td>065</td>
<td>066</td>
<td>067</td>
<td>068</td>
<td>069</td>
</tr>
<tr>
<td>07_</td>
<td>070</td>
<td>071</td>
<td>072</td>
<td>073</td>
<td>074</td>
<td>075</td>
<td>076</td>
<td>077</td>
<td>078</td>
<td>079</td>
</tr>
<tr>
<td>08_</td>
<td>080</td>
<td>081</td>
<td>082</td>
<td>083</td>
<td>084</td>
<td>085</td>
<td>086</td>
<td>087</td>
<td>088</td>
<td>089</td>
</tr>
<tr>
<td>09_</td>
<td>090</td>
<td>091</td>
<td>092</td>
<td>093</td>
<td>094</td>
<td>095</td>
<td>096</td>
<td>097</td>
<td>098</td>
<td>099</td>
</tr>
<tr>
<td>10_</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
<td>108</td>
<td>109</td>
</tr>
<tr>
<td>11_</td>
<td>110</td>
<td>111</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
<td>118</td>
<td>119</td>
</tr>
<tr>
<td>12_</td>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
<td>127</td>
<td>128</td>
<td>129</td>
</tr>
<tr>
<td>13_</td>
<td>130</td>
<td>131</td>
<td>132</td>
<td>133</td>
<td>134</td>
<td>135</td>
<td>136</td>
<td>137</td>
<td>138</td>
<td>139</td>
</tr>
<tr>
<td>14_</td>
<td>140</td>
<td>141</td>
<td>142</td>
<td>143</td>
<td>144</td>
<td>145</td>
<td>146</td>
<td>147</td>
<td>148</td>
<td>149</td>
</tr>
<tr>
<td>15_</td>
<td>150</td>
<td>151</td>
<td>152</td>
<td>153</td>
<td>154</td>
<td>155</td>
<td>156</td>
<td>157</td>
<td>158</td>
<td>159</td>
</tr>
<tr>
<td>16_</td>
<td>160</td>
<td>161</td>
<td>162</td>
<td>163</td>
<td>164</td>
<td>165</td>
<td>166</td>
<td>167</td>
<td>168</td>
<td>169</td>
</tr>
<tr>
<td>17_</td>
<td>170</td>
<td>171</td>
<td>172</td>
<td>173</td>
<td>174</td>
<td>175</td>
<td>176</td>
<td>177</td>
<td>178</td>
<td>179</td>
</tr>
<tr>
<td>18_</td>
<td>180</td>
<td>181</td>
<td>182</td>
<td>183</td>
<td>184</td>
<td>185</td>
<td>186</td>
<td>187</td>
<td>188</td>
<td>189</td>
</tr>
<tr>
<td>19_</td>
<td>190</td>
<td>191</td>
<td>192</td>
<td>193</td>
<td>194</td>
<td>195</td>
<td>196</td>
<td>197</td>
<td>198</td>
<td>199</td>
</tr>
<tr>
<td>20_</td>
<td>200</td>
<td>201</td>
<td>202</td>
<td>203</td>
<td>204</td>
<td>205</td>
<td>206</td>
<td>207</td>
<td>208</td>
<td>209</td>
</tr>
<tr>
<td>21_</td>
<td>210</td>
<td>211</td>
<td>212</td>
<td>213</td>
<td>214</td>
<td>215</td>
<td>216</td>
<td>217</td>
<td>218</td>
<td>219</td>
</tr>
<tr>
<td>22_</td>
<td>220</td>
<td>221</td>
<td>222</td>
<td>223</td>
<td>224</td>
<td>225</td>
<td>226</td>
<td>227</td>
<td>228</td>
<td>229</td>
</tr>
<tr>
<td>23_</td>
<td>230</td>
<td>231</td>
<td>232</td>
<td>233</td>
<td>234</td>
<td>235</td>
<td>236</td>
<td>237</td>
<td>238</td>
<td>239</td>
</tr>
<tr>
<td>24_</td>
<td>240</td>
<td>241</td>
<td>242</td>
<td>243</td>
<td>244</td>
<td>245</td>
<td>246</td>
<td>247</td>
<td>248</td>
<td>249</td>
</tr>
<tr>
<td>25_</td>
<td>250</td>
<td>251</td>
<td>252</td>
<td>253</td>
<td>254</td>
<td>255</td>
<td>256</td>
<td>257</td>
<td>258</td>
<td>259</td>
</tr>
<tr>
<td>26_</td>
<td>260</td>
<td>261</td>
<td>262</td>
<td>263</td>
<td>264</td>
<td>265</td>
<td>266</td>
<td>267</td>
<td>268</td>
<td>269</td>
</tr>
<tr>
<td>27_</td>
<td>270</td>
<td>271</td>
<td>272</td>
<td>273</td>
<td>274</td>
<td>275</td>
<td>276</td>
<td>277</td>
<td>278</td>
<td>279</td>
</tr>
<tr>
<td>28_</td>
<td>280</td>
<td>281</td>
<td>282</td>
<td>283</td>
<td>284</td>
<td>285</td>
<td>286</td>
<td>287</td>
<td>288</td>
<td>289</td>
</tr>
<tr>
<td>29_</td>
<td>290</td>
<td>291</td>
<td>292</td>
<td>293</td>
<td>294</td>
<td>295</td>
<td>296</td>
<td>297</td>
<td>298</td>
<td>299</td>
</tr>
<tr>
<td>30_</td>
<td>300</td>
<td>301</td>
<td>302</td>
<td>303</td>
<td>304</td>
<td>305</td>
<td>306</td>
<td>307</td>
<td>308</td>
<td>309</td>
</tr>
<tr>
<td>31_</td>
<td>310</td>
<td>311</td>
<td>312</td>
<td>313</td>
<td>314</td>
<td>315</td>
<td>316</td>
<td>317</td>
<td>318</td>
<td>319</td>
</tr>
<tr>
<td>32_</td>
<td>320</td>
<td>321</td>
<td>322</td>
<td>323</td>
<td>324</td>
<td>325</td>
<td>326</td>
<td>327</td>
<td>328</td>
<td>329</td>
</tr>
<tr>
<td>33_</td>
<td>330</td>
<td>331</td>
<td>332</td>
<td>333</td>
<td>334</td>
<td>335</td>
<td>336</td>
<td>337</td>
<td>338</td>
<td>339</td>
</tr>
<tr>
<td>34_</td>
<td>340</td>
<td>341</td>
<td>342</td>
<td>343</td>
<td>344</td>
<td>345</td>
<td>346</td>
<td>347</td>
<td>348</td>
<td>349</td>
</tr>
<tr>
<td>35_</td>
<td>350</td>
<td>351</td>
<td>352</td>
<td>353</td>
<td>354</td>
<td>355</td>
<td>356</td>
<td>357</td>
<td>358</td>
<td>359</td>
</tr>
<tr>
<td>36_</td>
<td>360</td>
<td>361</td>
<td>362</td>
<td>363</td>
<td>364</td>
<td>365</td>
<td>366</td>
<td>367</td>
<td>368</td>
<td>369</td>
</tr>
<tr>
<td>37_</td>
<td>370</td>
<td>371</td>
<td>372</td>
<td>373</td>
<td>374</td>
<td>375</td>
<td>376</td>
<td>377</td>
<td>378</td>
<td>379</td>
</tr>
<tr>
<td>38_</td>
<td>380</td>
<td>381</td>
<td>382</td>
<td>383</td>
<td>384</td>
<td>385</td>
<td>386</td>
<td>387</td>
<td>388</td>
<td>389</td>
</tr>
<tr>
<td>39_</td>
<td>390</td>
<td>391</td>
<td>392</td>
<td>393</td>
<td>394</td>
<td>395</td>
<td>396</td>
<td>397</td>
<td>398</td>
<td>399</td>
</tr>
<tr>
<td>40_</td>
<td>400</td>
<td>401</td>
<td>402</td>
<td>403</td>
<td>404</td>
<td>405</td>
<td>406</td>
<td>407</td>
<td>408</td>
<td>409</td>
</tr>
<tr>
<td>41_</td>
<td>410</td>
<td>411</td>
<td>412</td>
<td>413</td>
<td>414</td>
<td>415</td>
<td>416</td>
<td>417</td>
<td>418</td>
<td>419</td>
</tr>
<tr>
<td>42_</td>
<td>420</td>
<td>421</td>
<td>422</td>
<td>423</td>
<td>424</td>
<td>425</td>
<td>426</td>
<td>427</td>
<td>428</td>
<td>429</td>
</tr>
<tr>
<td>43_</td>
<td>430</td>
<td>431</td>
<td>432</td>
<td>433</td>
<td>434</td>
<td>435</td>
<td>436</td>
<td>437</td>
<td>438</td>
<td>439</td>
</tr>
<tr>
<td>44_</td>
<td>440</td>
<td>441</td>
<td>442</td>
<td>443</td>
<td>444</td>
<td>445</td>
<td>446</td>
<td>447</td>
<td>448</td>
<td>449</td>
</tr>
<tr>
<td>45_</td>
<td>450</td>
<td>451</td>
<td>452</td>
<td>453</td>
<td>454</td>
<td>455</td>
<td>456</td>
<td>457</td>
<td>458</td>
<td>459</td>
</tr>
<tr>
<td>46_</td>
<td>460</td>
<td>461</td>
<td>462</td>
<td>463</td>
<td>464</td>
<td>465</td>
<td>466</td>
<td>467</td>
<td>468</td>
<td>469</td>
</tr>
<tr>
<td>47_</td>
<td>470</td>
<td>471</td>
<td>472</td>
<td>473</td>
<td>474</td>
<td>475</td>
<td>476</td>
<td>477</td>
<td>478</td>
<td>479</td>
</tr>
<tr>
<td>48_</td>
<td>480</td>
<td>481</td>
<td>482</td>
<td>483</td>
<td>484</td>
<td>485</td>
<td>486</td>
<td>487</td>
<td>488</td>
<td>489</td>
</tr>
<tr>
<td>49_</td>
<td>490</td>
<td>491</td>
<td>492</td>
<td>493</td>
<td>494</td>
<td>495</td>
<td>496</td>
<td>497</td>
<td>498</td>
<td>499</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>001</td>
<td>002</td>
<td>003</td>
<td>004</td>
<td>005</td>
<td>006</td>
<td>007</td>
<td>008</td>
<td>009</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>010</td>
<td>011</td>
<td>012</td>
<td>013</td>
<td>014</td>
<td>015</td>
<td>016</td>
<td>017</td>
<td>018</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>020</td>
<td>021</td>
<td>022</td>
<td>023</td>
<td>024</td>
<td>025</td>
<td>026</td>
<td>027</td>
<td>028</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>030</td>
<td>031</td>
<td>032</td>
<td>033</td>
<td>034</td>
<td>035</td>
<td>036</td>
<td>037</td>
<td>038</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>040</td>
<td>041</td>
<td>042</td>
<td>043</td>
<td>044</td>
<td>045</td>
<td>046</td>
<td>047</td>
<td>048</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>050</td>
<td>051</td>
<td>052</td>
<td>053</td>
<td>054</td>
<td>055</td>
<td>056</td>
<td>057</td>
<td>058</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>060</td>
<td>061</td>
<td>062</td>
<td>063</td>
<td>064</td>
<td>065</td>
<td>066</td>
<td>067</td>
<td>068</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>070</td>
<td>071</td>
<td>072</td>
<td>073</td>
<td>074</td>
<td>075</td>
<td>076</td>
<td>077</td>
<td>078</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
<td>108</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>110</td>
<td>111</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
<td>118</td>
</tr>
<tr>
<td>A</td>
<td>0A</td>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
<td>127</td>
<td>128</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>130</td>
<td>131</td>
<td>132</td>
<td>133</td>
<td>134</td>
<td>135</td>
<td>136</td>
<td>137</td>
<td>138</td>
</tr>
<tr>
<td>C</td>
<td>0C</td>
<td>140</td>
<td>141</td>
<td>142</td>
<td>143</td>
<td>144</td>
<td>145</td>
<td>146</td>
<td>147</td>
<td>148</td>
</tr>
<tr>
<td>D</td>
<td>0D</td>
<td>150</td>
<td>151</td>
<td>152</td>
<td>153</td>
<td>154</td>
<td>155</td>
<td>156</td>
<td>157</td>
<td>158</td>
</tr>
<tr>
<td>E</td>
<td>0E</td>
<td>160</td>
<td>161</td>
<td>162</td>
<td>163</td>
<td>164</td>
<td>165</td>
<td>166</td>
<td>167</td>
<td>168</td>
</tr>
<tr>
<td>F</td>
<td>0F</td>
<td>170</td>
<td>171</td>
<td>172</td>
<td>173</td>
<td>174</td>
<td>175</td>
<td>176</td>
<td>177</td>
<td>178</td>
</tr>
</tbody>
</table>

Table 147: DPD-to-BCD translation
Appendix C. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided that defines simple shorthand for the most frequently used forms of Branch Conditional, Compare, Trap, Rotate and Shift, and certain other instructions.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

C.1 Symbols

The following symbols are defined for use in instructions (basic or extended mnemonics) that specify a Condition Register field or a Condition Register bit. The first five (lt, ..., un) identify a bit number within a CR field. The remainder (cr0, ..., cr7) identify a CR field. An expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol and 32 can be used to identify a CR bit.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>0</td>
<td>Less than</td>
</tr>
<tr>
<td>gt</td>
<td>1</td>
<td>Greater than</td>
</tr>
<tr>
<td>eq</td>
<td>2</td>
<td>Equal</td>
</tr>
<tr>
<td>so</td>
<td>3</td>
<td>Summary overflow</td>
</tr>
<tr>
<td>un</td>
<td>3</td>
<td>Unordered (after floating-point comparison)</td>
</tr>
<tr>
<td>cr0</td>
<td>0</td>
<td>CR Field 0</td>
</tr>
<tr>
<td>cr1</td>
<td>1</td>
<td>CR Field 1</td>
</tr>
<tr>
<td>cr2</td>
<td>2</td>
<td>CR Field 2</td>
</tr>
<tr>
<td>cr3</td>
<td>3</td>
<td>CR Field 3</td>
</tr>
<tr>
<td>cr4</td>
<td>4</td>
<td>CR Field 4</td>
</tr>
<tr>
<td>cr5</td>
<td>5</td>
<td>CR Field 5</td>
</tr>
<tr>
<td>cr6</td>
<td>6</td>
<td>CR Field 6</td>
</tr>
<tr>
<td>cr7</td>
<td>7</td>
<td>CR Field 7</td>
</tr>
</tbody>
</table>

The extended mnemonics in Sections C.2.2 and C.3 require identification of a CR bit: if one of the CR field symbols is used, it must be multiplied by 4 and added to a bit-number-within-CR-field (value in the range 0-3, explicit or symbolic) and 32. The extended mnemonics in Sections C.2.3 and C.5 require identification of a CR field: if one of the CR field symbols is used, it must not be multiplied by 4 or added to 32. (For the extended mnemonics in Section C.2.3, the bit number within the CR field is part of the extended mnemonic. The programmer identifies the CR field, and the Assembler does the multiplication and addition required to produce a CR bit number for the BI field of the underlying basic mnemonic.)
C.2 Branch Mnemonics

The mnemonics discussed in this section are variations of the Branch Conditional instructions.

Note: bclr, bclrl, bcctr, and bcctrl each serve as both a basic and an extended mnemonic. The Assembler will recognize a bclr, bclrl, bcctr, or bcctrl mnemonic with three operands as the basic form, and a bclr, bclrl, bcctr, or bcctrl mnemonic with two operands as the extended form. In the extended form the BH operand is omitted and assumed to be 0b00. Similarly, for all the extended mnemonics described in Sections C.2.2 - C.2.4 that devolve to any of these four basic mnemonics the BH operand can either be coded or omitted. If it is omitted it is assumed to be 0b00.

C.2.1 BO and BI Fields

The 5-bit BO and BI fields control whether the branch is taken. Providing an extended mnemonic for every possible combination of these fields would be neither useful nor practical. The mnemonics described in Sections C.2.2 - C.2.4 include the most useful cases. Other cases can be coded using a basic Branch Conditional mnemonic (bc[l][a], bclrl[ ], bcctr[ ]) with the appropriate operands.

C.2.2 Simple Branch Mnemonics

Instructions using one of the mnemonics in Table 148 that tests a Condition Register bit specify the corresponding bit as the first operand. The symbols defined in Section C.1 can be used in this operand.

Notice that there are no extended mnemonics for relative and absolute unconditional branches. For these the basic mnemonics b, ba, bl, and bla should be used.

Table 148: Simple branch mnemonics

<table>
<thead>
<tr>
<th>Branch Semantics</th>
<th>LR not Set</th>
<th>LR Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bc Relative</td>
<td>bca Absolute</td>
</tr>
<tr>
<td>Branch unconditionally</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Branch if CR[BI]=1</td>
<td>bt</td>
<td>bta</td>
</tr>
<tr>
<td>Branch if CR[BI]=0</td>
<td>bf</td>
<td>bfa</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR nonzero</td>
<td>bdnz</td>
<td>bdnza</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR nonzero and CR[BI]=1</td>
<td>bdnzt</td>
<td>bdnzta</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR nonzero and CR[BI]=0</td>
<td>bdnzf</td>
<td>bdnzfa</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR zero</td>
<td>bdz</td>
<td>bdza</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR zero and CR[BI]=1</td>
<td>bdzt</td>
<td>bdzta</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR zero and CR[BI]=0</td>
<td>bdzf</td>
<td>bdzfa</td>
</tr>
</tbody>
</table>

Examples

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR).
   
   bdnz target  
   (equivalent to: bc 16,0,target)

2. Same as (1) but branch only if CTR is nonzero and condition in CR0 is "equal".
   
   bdnzt eq,target  
   (equivalent to: bc 8,2,target)

3. Same as (2), but "equal" condition is in CR5.
   
   bdnzt 4×cr5+eq,target  
   (equivalent to: bc 8,22,target)
4. Branch if bit 59 of CR is 0.
   bf 27,target  (equivalent to:  bc 4,27,target)

5. Same as (4), but set the Link Register. This is a form of conditional “call”.
   bfl 27,target  (equivalent to: bcl 4,27,target)

C.2.3 Branch Mnemonics Incorporating Conditions

In the mnemonics defined in Table 149, the test of a bit in a Condition Register field is encoded in the mnemonic.

Instructions using the mnemonics in Table 149 specify the CR field as an optional first operand. One of the CR field symbols defined in Section C.1 can be used for this operand. If the CR field being tested is CR Field 0, this operand need not be specified unless the resulting basic mnemonic is \texttt{bclr} or \texttt{bcctr} and the BH operand is specified.

A standard set of codes has been adopted for the most common combinations of branch conditions.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>Less than</td>
</tr>
<tr>
<td>le</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>eq</td>
<td>Equal</td>
</tr>
<tr>
<td>ge</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>gt</td>
<td>Greater than</td>
</tr>
<tr>
<td>nl</td>
<td>Not less than</td>
</tr>
<tr>
<td>ne</td>
<td>Not equal</td>
</tr>
<tr>
<td>ng</td>
<td>Not greater than</td>
</tr>
<tr>
<td>so</td>
<td>Summary overflow</td>
</tr>
<tr>
<td>ns</td>
<td>Not summary overflow</td>
</tr>
<tr>
<td>un</td>
<td>Unordered (after floating-point comparison)</td>
</tr>
<tr>
<td>nu</td>
<td>Not unordered (after floating-point comparison)</td>
</tr>
</tbody>
</table>

These codes are reflected in the mnemonics shown in Table 149.

<table>
<thead>
<tr>
<th>Branch Semantics</th>
<th>LR not Set</th>
<th>LR Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bc Relative</td>
<td>bca Absolute</td>
</tr>
<tr>
<td>Branch if less than</td>
<td>bit</td>
<td>bitla</td>
</tr>
<tr>
<td>Branch if less than or equal</td>
<td>ble</td>
<td>blea</td>
</tr>
<tr>
<td>Branch if equal</td>
<td>beq</td>
<td>beqa</td>
</tr>
<tr>
<td>Branch if greater than or equal</td>
<td>bgp</td>
<td>bgea</td>
</tr>
<tr>
<td>Branch if greater than</td>
<td>bgt</td>
<td>bgtla</td>
</tr>
<tr>
<td>Branch if not less than</td>
<td>bnl</td>
<td>bna</td>
</tr>
<tr>
<td>Branch if not equal</td>
<td>bne</td>
<td>bnea</td>
</tr>
<tr>
<td>Branch if not greater than</td>
<td>bng</td>
<td>bnga</td>
</tr>
<tr>
<td>Branch if summary overflow</td>
<td>bso</td>
<td>bsoa</td>
</tr>
<tr>
<td>Branch if not summary overflow</td>
<td>bns</td>
<td>bnsa</td>
</tr>
<tr>
<td>Branch if unordered</td>
<td>bun</td>
<td>buna</td>
</tr>
<tr>
<td>Branch if not unordered</td>
<td>bnu</td>
<td>bunua</td>
</tr>
</tbody>
</table>

Examples

1. Branch if CR0 reflects condition "not equal".
   \texttt{bne \ target}  (equivalent to:  bc 4,2,target)

2. Same as (1), but condition is in CR3.
bne cr3,target (equivalent to: bc 4,14,target)

3. Branch to an absolute target if CR4 specifies “greater than”, setting the Link Register. This is a form of conditional “call”.
   bgta cr4,target (equivalent to: bcla 12,17,target)

4. Same as (3), but target address is in the Count Register.
   bgtcrn cr4 (equivalent to: bcctnl 12,17,0)

C.2.4 Branch Prediction

Software can use the “at” bits of Branch Conditional instructions to provide a hint to the processor about the behavior of the branch. If, for a given such instruction, the branch is almost always taken or almost always not taken, a suffix can be added to the mnemonic indicating the value to be used for the “at” bits.

+ Predict branch to be taken (at=0b11)
- Predict branch not to be taken (at=0b10)

Such a suffix can be added to any Branch Conditional mnemonic, either basic or extended, that tests either the Count Register or a CR bit (but not both). Assemblers should use 0b00 as the default value for the “at” bits, indicating that software has offered no prediction.

Examples

1. Branch if CR0 reflects condition “less than”, specifying that the branch should be predicted to be taken.
   blt+ target

2. Same as (1), but target address is in the Link Register and the branch should be predicted not to be taken.
   blttr-
C.3 Condition Register Logical Mnemonics

The Condition Register Logical instructions can be used to set (to 1), clear (to 0), copy, or invert a given Condition Register bit. Extended mnemonics are provided that allow these operations to be coded easily.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition Register set</td>
<td>crset bx</td>
<td>creqv bx,bx,bx</td>
</tr>
<tr>
<td>Condition Register clear</td>
<td>crclr bx</td>
<td>crxor bx,bx,bx</td>
</tr>
<tr>
<td>Condition Register move</td>
<td>crmove bx,by</td>
<td>cror bx,by,by</td>
</tr>
<tr>
<td>Condition Register not</td>
<td>cmnot bx,by</td>
<td>cmor bx,by,by</td>
</tr>
</tbody>
</table>

The symbols defined in Section C.1 can be used to identify the Condition Register bits.

Examples
1. Set CR bit 57.
   
   crset 25 (equivalent to: creqv 25,25,25)

2. Clear the SO bit of CR0.
   
   crclr so (equivalent to: crxor 3,3,3)

3. Same as (2), but SO bit to be cleared is in CR3.
   
   crclr 4×cr3+so (equivalent to: crxor 15,15,15)

4. Invert the EQ bit.
   
   cmnot eq,eq (equivalent to: cmor 2,2,2)

5. Same as (4), but EQ bit to be inverted is in CR4, and the result is to be placed into the EQ bit of CR5.
   
   cmnot 4×cr5+eq,4×cr4+eq (equivalent to: cmor 22,18,18)

C.4 Subtract Mnemonics

C.4.1 Subtract Immediate

Although there is no "Subtract Immediate" instruction, its effect can be achieved by using an Add Immediate instruction with the immediate operand negated. Extended mnemonics are provided that include this negation, making the intent of the computation clearer.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>subi</td>
<td>Rx,Ry,value</td>
<td>addi Rx,Ry,-value</td>
</tr>
<tr>
<td>subis</td>
<td>Rx,Ry,value</td>
<td>addis Rx,Ry,-value</td>
</tr>
<tr>
<td>subic</td>
<td>Rx,Ry,value</td>
<td>addic Rx,Ry,-value</td>
</tr>
<tr>
<td>subic.</td>
<td>Rx,Ry,value</td>
<td>addic. Rx,Ry,-value</td>
</tr>
</tbody>
</table>

C.4.2 Subtract

The Subtract From instructions subtract the second operand (RA) from the third (RB). Extended mnemonics are provided that use the more "normal" order, in which the third operand is subtracted from the second. Both these mnemonics can be coded with a final "o" and/or "." to cause the OE and/or Rc bit to be set in the underlying instruction.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>Rx,Ry,Rz</td>
</tr>
<tr>
<td>subc</td>
<td>Rx,Ry,Rz</td>
</tr>
</tbody>
</table>
C.5 Compare Mnemonics

The L field in the fixed-point Compare instructions controls whether the operands are treated as 64-bit quantities or as 32-bit quantities. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

The BF field can be omitted if the result of the comparison is to be placed into CR Field 0. Otherwise the target CR field must be specified as the first operand. One of the CR field symbols defined in Section C.1 can be used for this operand.

Note: The Assembler will recognize a basic Compare mnemonic with three operands, and will generate the instruction with L=0. Thus the Assembler must require that the BF field, which normally can be omitted when CR Field 0 is the target, be specified explicitly if L is.

C.5.1 Doubleword Comparisons

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare doubleword immediate</td>
<td>cmpdi bf,ra,si</td>
<td>cmpi 1,ra,si</td>
</tr>
<tr>
<td>Compare doubleword</td>
<td>cmpd bf,ra,rb</td>
<td>cmp 1,ra,rb</td>
</tr>
<tr>
<td>Compare logical doubleword immediate</td>
<td>cmpldi bf,ra,ui</td>
<td>cmpli 1,ra,ui</td>
</tr>
<tr>
<td>Compare logical doubleword</td>
<td>cmpld bf,ra,rb</td>
<td>cmpl 1,ra,rb</td>
</tr>
</tbody>
</table>

Examples
1. Compare register Rx and immediate value 100 as unsigned 64-bit integers and place result into CR0.
   ```
   cmpldi Rx,100  
   ```
   (equivalent to:
   ```
   cmpi 0,1,Rx,100
   ```
2. Same as (1), but place result into CR4.
   ```
   cmpldi cr4,Rx,100
   ```
   (equivalent to:
   ```
   cmpli 4,1,Rx,100
   ```
3. Compare registers Rx and Ry as signed 64-bit integers and place result into CR0.
   ```
   cmpd Rx,Ry
   ```
   (equivalent to:
   ```
   cmp 0,1,Rx,Ry
   ```

C.5.2 Word Comparisons

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare word immediate</td>
<td>cmpwi bf,ra,si</td>
<td>cmpi 0,ra,si</td>
</tr>
<tr>
<td>Compare word</td>
<td>cmpw bf,ra,rb</td>
<td>cmp 0,ra,rb</td>
</tr>
<tr>
<td>Compare logical word immediate</td>
<td>cmplwi bf,ra,ui</td>
<td>cmpli 0,ra,ui</td>
</tr>
<tr>
<td>Compare logical word</td>
<td>cmplw bf,ra,rb</td>
<td>cmpl 0,ra,rb</td>
</tr>
</tbody>
</table>

Examples
1. Compare bits 32:63 of register Rx and immediate value 100 as signed 32-bit integers and place result into CR0.
   ```
   cmpwi Rx,100
   ```
   (equivalent to:
   ```
   cmpi 0,0,Rx,100
   ```
2. Same as (1), but place result into CR4.
   ```
   cmpwi cr4,Rx,100
   ```
   (equivalent to:
   ```
   cmpli 4,0,Rx,100
   ```
3. Compare bits 32:63 of registers Rx and Ry as unsigned 32-bit integers and place result into CR0.
   ```
   cmplw Rx,Ry
   ```
   (equivalent to:
   ```
   cmp 0,0,Rx,Ry
   ```
C.6 Trap Mnemonics

The mnemonics defined in Table 153 are variations of the Trap instructions, with the most useful values of TO represented in the mnemonic rather than specified as a numeric operand.

A standard set of codes has been adopted for the most common combinations of trap conditions.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>TO encoding</th>
<th>&lt;</th>
<th>&gt;</th>
<th>=</th>
<th>&lt;=u</th>
<th>&gt;=u</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>Less than</td>
<td>16</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>le</td>
<td>Less than or equal</td>
<td>20</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>eq</td>
<td>Equal</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ge</td>
<td>Greater than or equal</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gt</td>
<td>Greater than</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nl</td>
<td>Not less than</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ne</td>
<td>Not equal</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ng</td>
<td>Not greater than</td>
<td>20</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>llt</td>
<td>Logically less than</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lle</td>
<td>Logically less than or equal</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lge</td>
<td>Logically greater than or equal</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lgt</td>
<td>Logically greater than</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nlt</td>
<td>Logically not less than</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lng</td>
<td>Logically not greater than</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>u</td>
<td>Unconditionally with parameters</td>
<td>31</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(none)</td>
<td>Unconditional</td>
<td>31</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

These codes are reflected in the mnemonics shown in Table 153.

Table 153: Trap mnemonics

<table>
<thead>
<tr>
<th>Trap Semantics</th>
<th>64-bit Comparison</th>
<th>32-bit Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tdi Immediate</td>
<td>td Register</td>
</tr>
<tr>
<td>Trap unconditionally</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Trap unconditionally with parameters</td>
<td>tdui</td>
<td>tdu</td>
</tr>
<tr>
<td>Trap if less than</td>
<td>tdliti</td>
<td>tdlit</td>
</tr>
<tr>
<td>Trap if less than or equal</td>
<td>tdliei</td>
<td>tdle</td>
</tr>
<tr>
<td>Trap if equal</td>
<td>tdeqi</td>
<td>tdeq</td>
</tr>
<tr>
<td>Trap if greater than or equal</td>
<td>tdgei</td>
<td>tdge</td>
</tr>
<tr>
<td>Trap if greater than</td>
<td>tdgti</td>
<td>tdgt</td>
</tr>
<tr>
<td>Trap if not less than</td>
<td>tdlili</td>
<td>tdlil</td>
</tr>
<tr>
<td>Trap if not equal</td>
<td>tdei</td>
<td>tdne</td>
</tr>
<tr>
<td>Trap if not greater than</td>
<td>tdnji</td>
<td>tdnj</td>
</tr>
<tr>
<td>Trap if logically less than</td>
<td>tdltli</td>
<td>tdllt</td>
</tr>
<tr>
<td>Trap if logically less than or equal</td>
<td>tdliei</td>
<td>tdlle</td>
</tr>
<tr>
<td>Trap if logically greater than or equal</td>
<td>tdgei</td>
<td>tdlge</td>
</tr>
<tr>
<td>Trap if logically greater than</td>
<td>tdgti</td>
<td>tdgt</td>
</tr>
<tr>
<td>Trap if logically not less than</td>
<td>tdlini</td>
<td>tdlin</td>
</tr>
<tr>
<td>Trap if logically not greater than</td>
<td>tdlgi</td>
<td>tdlng</td>
</tr>
</tbody>
</table>

Table 153: Trap mnemonics

<table>
<thead>
<tr>
<th>Trap Semantics</th>
<th>64-bit Comparison</th>
<th>32-bit Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>twi Immediate</td>
<td>tw Register</td>
</tr>
<tr>
<td>Trap unconditionally</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Trap unconditionally with parameters</td>
<td>twui</td>
<td>twu</td>
</tr>
<tr>
<td>Trap if less than</td>
<td>twliti</td>
<td>twliti</td>
</tr>
<tr>
<td>Trap if less than or equal</td>
<td>twliei</td>
<td>twle</td>
</tr>
<tr>
<td>Trap if equal</td>
<td>twdeqi</td>
<td>twde</td>
</tr>
<tr>
<td>Trap if greater than or equal</td>
<td>twdgei</td>
<td>twge</td>
</tr>
<tr>
<td>Trap if greater than</td>
<td>twgti</td>
<td>twgt</td>
</tr>
<tr>
<td>Trap if not less than</td>
<td>twlili</td>
<td>twlil</td>
</tr>
<tr>
<td>Trap if not equal</td>
<td>twnei</td>
<td>twne</td>
</tr>
<tr>
<td>Trap if not greater than</td>
<td>twngi</td>
<td>twng</td>
</tr>
<tr>
<td>Trap if logically less than</td>
<td>twliti</td>
<td>twlitt</td>
</tr>
<tr>
<td>Trap if logically less than or equal</td>
<td>twliei</td>
<td>twlile</td>
</tr>
<tr>
<td>Trap if logically greater than or equal</td>
<td>twdgei</td>
<td>twlge</td>
</tr>
<tr>
<td>Trap if logically greater than</td>
<td>twgti</td>
<td>twgt</td>
</tr>
<tr>
<td>Trap if logically not less than</td>
<td>twlili</td>
<td>twlin</td>
</tr>
<tr>
<td>Trap if logically not greater than</td>
<td>twlgi</td>
<td>twling</td>
</tr>
</tbody>
</table>

Table 153: Trap mnemonics
Examples
1. Trap if register Rx is not 0.
   tdnei Rx,0 (equivalent to: tdi 24,Rx,0)
2. Same as (1), but comparison is to register Ry.
   tdne Rx,Ry (equivalent to: td 24,Rx,Ry)
3. Trap if bits 32:63 of register Rx, considered as a 32-bit quantity, are logically greater than 0x7FF.
   twlgti Rx,0x7FF (equivalent to: twi 1,Rx,0x7FF)
4. Trap unconditionally.
   trap (equivalent to: tw 31,0,0)
5. Trap unconditionally with immediate parameters Rx and Ry
   tdu Rx,Ry (equivalent to: td 31,Rx,Ry)

C.7 Integer Select Mnemonics

The mnemonics defined in Table 154, “Integer Select mnemonics,” on page 1018 are variations of the Integer Select instructions, with the most useful values of BC represented in the mnemonic rather than specified as a numeric operand.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>Less than</td>
</tr>
<tr>
<td>eq</td>
<td>Equal</td>
</tr>
<tr>
<td>gt</td>
<td>Greater than</td>
</tr>
</tbody>
</table>

These codes are reflected in the mnemonics shown in Table 154.

<table>
<thead>
<tr>
<th>Table 154: Integer Select mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Select semantics</strong></td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>Integer Select if less than</td>
</tr>
<tr>
<td>Integer Select if equal</td>
</tr>
<tr>
<td>Integer Select if greater than</td>
</tr>
</tbody>
</table>

Examples
1. Set register Rx to Ry if the LT bit is set in CR0, and to Rz otherwise.
   isellt Rx,Ry,Rz (equivalent to: isel Rx,Ry,Rz,0)
2. Set register Rx to Ry if the GT bit is set in CR0, and to Rz otherwise.
   iselgt Rx,Ry,Rz (equivalent to: isel Rx,Ry,Rz,1)
3. Set register Rx to Ry if the EQ bit is set in CR0, and to Rz otherwise.
   iseleq Rx,Ry,Rz (equivalent to: isel Rx,Ry,Rz,2)
C.8 Rotate and Shift Mnemonics

The Rotate and Shift instructions provide powerful and general ways to manipulate register contents, but can be difficult to understand. Extended mnemonics are provided that allow some of the simpler operations to be coded easily.

Mnemonics are provided for the following types of operation.

- **Extract** Select a field of n bits starting at bit position b in the source register; left or right justify this field in the target register; clear all other bits of the target register to 0.

- **Insert** Select a left-justified or right-justified field of n bits in the source register; insert this field starting at bit position b of the target register; leave other bits of the target register unchanged. (No extended mnemonic is provided for insertion of a left-justified field when operating on doublewords, because such an insertion requires more than one instruction.)

- **Rotate** Rotate the contents of a register right or left n bits without masking.

- **Shift** Shift the contents of a register right or left n bits, clearing vacated bits to 0 (logical shift).

- **Clear** Clear the leftmost or rightmost n bits of a register to 0.

- **Clear left and shift left** Clear the leftmost b bits of a register, then shift the register left by n bits. This operation can be used to scale a (known nonnegative) array index by the width of an element.

C.8.1 Operations on Doublewords

All these mnemonics can be coded with a final “.” to cause the Rc bit to be set in the underlying instruction.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract and left justify immediate</td>
<td>extldi ra,rs,n,b (n &gt; 0)</td>
<td>rldicl ra,rs,b,n-1</td>
</tr>
<tr>
<td>Extract and right justify immediate</td>
<td>extrdi ra,rs,n,b (n &gt; 0)</td>
<td>rldicl ra,rs,b+n,64-n</td>
</tr>
<tr>
<td>Insert from right immediate</td>
<td>insrdi ra,rs,n,b (n &gt; 0)</td>
<td>rldimi ra,rs,64-(b+n),b</td>
</tr>
<tr>
<td>Rotate left immediate</td>
<td>rotldi ra,rs,n</td>
<td>rldicl ra,rs,n,0</td>
</tr>
<tr>
<td>Rotate right immediate</td>
<td>rotrdi ra,rs,n</td>
<td>rldicl ra,rs,64-n,0</td>
</tr>
<tr>
<td>Rotate left</td>
<td>rotld ra,rs,rb</td>
<td>rldcl ra,rs,rb,0</td>
</tr>
<tr>
<td>Shift left immediate</td>
<td>sldi ra,rs,n (n &lt; 64)</td>
<td>rldicl ra,rs,n,63-n</td>
</tr>
<tr>
<td>Shift right immediate</td>
<td>srdi ra,rs,n (n &lt; 64)</td>
<td>rldicl ra,rs,64-n,n</td>
</tr>
<tr>
<td>Clear left immediate</td>
<td>clrldi ra,rs,n (n &lt; 64)</td>
<td>rldicl ra,rs,0,n</td>
</tr>
<tr>
<td>Clear right immediate</td>
<td>clrrdi ra,rs,n (n &lt; 64)</td>
<td>rldicl ra,rs,0,63-n</td>
</tr>
<tr>
<td>Clear left and shift left immediate</td>
<td>clrldi ra,rs,b,n (n &lt;= b &lt; 64)</td>
<td>rldic ra,rs,n,b-n</td>
</tr>
</tbody>
</table>

Examples

1. Extract the sign bit (bit 0) of register Ry and place the result right-justified into register Rx.
   
   extrdi Rx,Ry,1,0 (equivalent to: rldicl Rx,Ry,1,63)

2. Insert the bit extracted in (1) into the sign bit (bit 0) of register Rz.
   
   insrdi Rz,Rx,1,0 (equivalent to: rldimi Rz,Rx,63,0)

3. Shift the contents of register Rx left 8 bits.
   
   sldi Rx,Rx,8 (equivalent to: rldicl Rx,Rx,8,55)

4. Clear the high-order 32 bits of register Ry and place the result into register Rx.
   
   clrrdi Rx,Ry,32 (equivalent to: rldicl Rx,Ry,0,32)
C.8.2 Operations on Words

All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction. The operations as described above apply to the low-order 32 bits of the registers, as if the registers were 32-bit registers. The Insert operations either preserve the high-order 32 bits of the target register or place rotated data there; the other operations clear these bits.

### Examples

1. Extract the sign bit (bit 32) of register Ry and place the result right-justified into register Rx.
   
   ```
   extrwi Rx,Ry,1,0  
   ```
   
   (equivalent to:
   ```
   rlwinm Ra,Rs,b,0,n-1
   ```

2. Insert the bit extracted in (1) into the sign bit (bit 32) of register Rz.
   
   ```
   insrwi Rz,Rx,1,0  
   ```
   
   (equivalent to: 
   ```
   rlwimi Ra,Rs,32-b,b,(b+n)-1
   ```

3. Shift the contents of register Rx left 8 bits, clearing the high-order 32 bits.
   
   ```
   slwi Rx,Rx,8  
   ```
   
   (equivalent to: 
   ```
   rlwinm Ra,Rs,0,0,31
   ```

4. Clear the high-order 16 bits of the low-order 32 bits of register Ry and place the result into register Rx, clearing the high-order 32 bits of register Rx.
   
   ```
   clrlwi Rx,Ry,16  
   ```
   
   (equivalent to:
   ```
   rlwinm Ra,Rs,b-n,31-n
   ```

---

### Table 156: Word rotate and shift mnemonics

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract and left justify immediate</td>
<td>extlwi ra,rs,n,b</td>
<td>rlwinm ra,rs,b,0,n-1</td>
</tr>
<tr>
<td>Extract and right justify immediate</td>
<td>extrwi ra,rs,n,b</td>
<td>rlwinm ra,rs,b+n,32-n,31</td>
</tr>
<tr>
<td>Insert from left immediate</td>
<td>inslwi ra,rs,n,b</td>
<td>rlwimi ra,rs,32-b,b,(b+n)-1</td>
</tr>
<tr>
<td>Insert from right immediate</td>
<td>insrwi ra,rs,n,b</td>
<td>rlwimi ra,rs,32-(b+n),b,(b+n)-1</td>
</tr>
<tr>
<td>Rotate left immediate</td>
<td>rotlwi ra,rs,n</td>
<td>rlwinm ra,rs,n,0,31</td>
</tr>
<tr>
<td>Rotate right immediate</td>
<td>rotlwi ra,rs,n</td>
<td>rlwinm ra,rs,32-n,0,31</td>
</tr>
<tr>
<td>Rotate left</td>
<td>rotlwi ra,rs,n</td>
<td>rlwinm ra,rs,rb,0,31</td>
</tr>
<tr>
<td>Shift left immediate</td>
<td>slwi ra,rs,n</td>
<td>rlwinm ra,rs,n,0,31-n</td>
</tr>
<tr>
<td>Shift right immediate</td>
<td>slrwi ra,rs,n</td>
<td>rlwinm ra,rs,32-n,n,31</td>
</tr>
<tr>
<td>Clear left immediate</td>
<td>clrlwi ra,rs,n</td>
<td>rlwinm ra,rs,0,n,31</td>
</tr>
<tr>
<td>Clear right immediate</td>
<td>clrrwi ra,rs,n</td>
<td>rlwinm ra,rs,0,0,31-n</td>
</tr>
<tr>
<td>Clear left and shift left immediate</td>
<td>clrlslwi ra,rs,b,n</td>
<td>rlwinm ra,rs,n,b-n,31-n</td>
</tr>
</tbody>
</table>
C.9 Move To/From Special Purpose Register Mnemonics

The `mtspr` and `mfspr` instructions specify a Special Purpose Register (SPR) as a numeric operand. Extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand.

<table>
<thead>
<tr>
<th>Special Purpose Register</th>
<th>Move To SPR</th>
<th>Move From SPR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Extended</td>
<td>Equivalent to</td>
</tr>
<tr>
<td>XER</td>
<td>mtvxer Rx</td>
<td>mtsprr 1,Rx</td>
</tr>
<tr>
<td>DSCR</td>
<td>mtudscr Rx</td>
<td>mtsprr 3,Rx</td>
</tr>
<tr>
<td>LR</td>
<td>mtlr Rx</td>
<td>mtsprr 8,Rx</td>
</tr>
<tr>
<td>CTR</td>
<td>mtctr Rx</td>
<td>mtsprr 9,Rx</td>
</tr>
<tr>
<td>AMR</td>
<td>mtamr Rx</td>
<td>mtsprr 13,Rx</td>
</tr>
<tr>
<td>CTRL</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VRSAVE</td>
<td>mtvrsave Rx</td>
<td>mtsprr 256,Rx</td>
</tr>
<tr>
<td>SPRG3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TB</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TBU</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SIER</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MMCRO</td>
<td>mtummcr2 Rx</td>
<td>mtsprr 769,Rx</td>
</tr>
<tr>
<td>MMCRA</td>
<td>mtummcr4 Rx</td>
<td>mtsprr 770,Rx</td>
</tr>
<tr>
<td>PMC1</td>
<td>mtumpc1 Rx</td>
<td>mtsprr 771,Rx</td>
</tr>
<tr>
<td>PMC2</td>
<td>mtumpc2 Rx</td>
<td>mtsprr 772,Rx</td>
</tr>
<tr>
<td>PMC3</td>
<td>mtumpc3 Rx</td>
<td>mtsprr 773,Rx</td>
</tr>
<tr>
<td>PMC4</td>
<td>mtumpc4 Rx</td>
<td>mtsprr 774,Rx</td>
</tr>
<tr>
<td>PMC5</td>
<td>mtumpc5 Rx</td>
<td>mtsprr 775,Rx</td>
</tr>
<tr>
<td>PMC6</td>
<td>mtumpc6 Rx</td>
<td>mtsprr 776,Rx</td>
</tr>
<tr>
<td>MCCR0</td>
<td>mtummcr0 Rx</td>
<td>mtsprr 779,Rx</td>
</tr>
<tr>
<td>SIAR</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SDAR</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MCCR1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BESCRS</td>
<td>mtbescrs Rx</td>
<td>mtsprr 800,Rx</td>
</tr>
<tr>
<td>BESCRU</td>
<td>mtbescru Rx</td>
<td>mtsprr 801,Rx</td>
</tr>
<tr>
<td>BESCRR</td>
<td>mtbescr Rx</td>
<td>mtsprr 802,Rx</td>
</tr>
<tr>
<td>BESCRRR</td>
<td>mtbescru Rx</td>
<td>mtsprr 803,Rx</td>
</tr>
<tr>
<td>EBHHR</td>
<td>mtebbhr Rx</td>
<td>mtsprr 804,Rx</td>
</tr>
<tr>
<td>EBBRR</td>
<td>mtebrr Rx</td>
<td>mtsprr 805,Rx</td>
</tr>
<tr>
<td>BESCR</td>
<td>mtbescr Rx</td>
<td>mtsprr 806,Rx</td>
</tr>
<tr>
<td>TAR</td>
<td>mttar Rx</td>
<td>mtsprr 815,Rx</td>
</tr>
<tr>
<td>PPR</td>
<td>mtppr Rx</td>
<td>mtsprr 896,Rx</td>
</tr>
<tr>
<td>PPR32</td>
<td>mtppr32 Rx</td>
<td>mtsprr 898,Rx</td>
</tr>
</tbody>
</table>

Examples

1. Copy the contents of register Rx to the XER.
   ```
   mtvxer Rx  
   ```

   (equivalent to: `mtspr 1,Rx`)
2. Copy the contents of the LR to register Rx.
   
   \texttt{mfr Rx} \quad \textit{(equivalent to: \texttt{mfsp}} \texttt{r Rx,8)}

3. Copy the contents of register Rx to the CTR.
   
   \texttt{mtctr Rx} \quad \textit{(equivalent to: \texttt{mtsp}} \texttt{r 9,Rx)}

\section*{C.10 Miscellaneous Mnemonics}

\subsection*{No-op}

Many Power ISA instructions can be coded in a way such that, effectively, no operation is performed. An extended mnemonic is provided for the preferred form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the no-op that will trigger this.

\texttt{nop} \quad \textit{(equivalent to: \texttt{ori}} \texttt{0,0,0)}

For some uses of a no-op instruction, optimizations related to no-ops, such as removal from the execution stream, are not desireable. An extended mnemonic is provided for the executed form of no-op. This form of no-op will still consume execution resources.

\texttt{xnop} \quad \textit{(equivalent to: \texttt{xori}} \texttt{0,0,0)}

To avoid certain security vulnerabilities, it is sometimes desirable to constrain the order in which instructions are executed at certain points in a program. An extended mnemonic is provided for a form of the \textit{Or Immediate} instruction that serves this purpose. See Section 5.4.3 of Book III.

\texttt{exser} \quad \textit{(equivalent to: \texttt{ori}} \texttt{31,31,0)}

\subsection*{Load Immediate}

The \textit{addi} and \textit{addis} instructions can be used to load an immediate value into a register. Extended mnemonics are provided to convey the idea that no addition is being performed but merely data movement (from the immediate field of the instruction to a register).

Load a 16-bit signed immediate value into register Rx.

\texttt{li Rx,value} \quad \textit{(equivalent to: \texttt{addi}} \texttt{Rx,0,value)}

Load a 16-bit signed immediate value, shifted left by 16 bits, into register Rx.

\texttt{lis Rx,value} \quad \textit{(equivalent to: \texttt{addis}} \texttt{Rx,0,value)}

\subsection*{Load Next Instruction Address}

The \textit{addpcis} instruction can be used to load the next instruction address into a register. An extended mnemonics is provided to perform this operation.

\texttt{lnia Rx} \quad \textit{(equivalent to: \texttt{addpcis}} \texttt{Rx,0)}
Load Address

This mnemonic permits computing the value of a base-displacement operand, using the `addi` instruction which normally requires separate register and immediate operands.

\[ \text{la Rx,D(Ry)} \quad \text{(equivalent to: addi Rx,Ry,D)} \]

The `la` mnemonic is useful for obtaining the address of a variable specified by name, allowing the Assembler to supply the base register number and compute the displacement. If the variable \( v \) is located at offset \( D_v \) bytes from the address in register \( R_v \), and the Assembler has been told to use register \( R_v \) as a base for references to the data structure containing \( v \), then the following line causes the address of \( v \) to be loaded into register \( Rx \).

\[ \text{la Rx,v} \quad \text{(equivalent to: addi Rx,Rv,Dv)} \]

Move Register

Several Power ISA instructions can be coded in a way such that they simply copy the contents of one register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register \( Ry \) to register \( Rx \). This mnemonic can be coded with a final "." to cause the \( Rc \) bit to be set in the underlying instruction.

\[ \text{mr Rx,Ry} \quad \text{(equivalent to: or Rx,Ry,Ry)} \]

Complement Register

Several Power ISA instructions can be coded in a way such that they complement the contents of one register and place the result into another register. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register \( Ry \) and places the result into register \( Rx \). This mnemonic can be coded with a final "." to cause the \( Rc \) bit to be set in the underlying instruction.

\[ \text{not Rx,Ry} \quad \text{(equivalent to: nor Rx,Ry,Ry)} \]

Move To/From Condition Register

This mnemonic permits copying the contents of the low-order 32 bits of a GPR to the Condition Register, using the same style as the `mfcr` instruction.

\[ \text{mtcr Rx} \quad \text{(equivalent to: mtcrf 0xFF,Rx)} \]

The following instructions may generate either the (old) `mtcrf` or `mfcr` instructions or the (new) `mtocrf` or `mfocrf` instruction, respectively, depending on the target machine type assembler parameter.

\[ \text{mtcrf FXM,Rx} \quad \text{mfcr Rx} \]

All three extended mnemonics in this subsection are being phased out. In future assemblers the form "mtcr Rx" may not exist, and the `mtcrf` and `mfcr` mnemonics may generate the old form instructions (with bit 11 = 0) regardless of the target machine type assembler parameter, or may cease to exist.
Book II:

Power ISA Virtual Environment Architecture
Chapter 1. Storage Model

1.1 Definitions

The following definitions, in addition to those specified in Book I, are used in this Book. In these definitions, "Load instruction" includes the Cache Management and other instructions that are stated in the instruction descriptions to be "treated as a Load", and similarly for "Store instruction".

- system
  A combination of processors, storage, and associated mechanisms that is capable of executing programs. Sometimes the reference to system includes services provided by the privileged software.

- main storage
  The level of storage hierarchy in which all storage state is visible to all processors and mechanisms in the system.

- normal memory
  Coherently-accessed, well-behaved regions of main storage that hold supervisor software and data and general purpose applications and data. This is in contrast with regions of main storage associated with accelerators or I/O interfaces or attached to other systems. Normal memory is assumed to have the following storage control attributes (see Section 1.6): not Write Through Required, not Caching Inhibited, Memory Coherence Required, and not Guarded.

- persistent storage
  Storage that retains its contents when power is removed, and is "behind" certain regions of main storage, that is not Caching Inhibited, in the sense that a store (to a location in such a region of main storage) will get to main storage before it gets to persistent storage.

- primary cache
  The level of cache closest to the processor.

- secondary cache
  After the primary cache, the next closest level of cache to the processor.

- instruction storage
  The view of storage as seen by the mechanism that fetches instructions.

- data storage
  The view of storage as seen by a Load or Store instruction.

- program order
  The execution of instructions in the order required by the sequential execution model. (See Section 2.2 of Book I.) A dcbz instruction that modifies storage which contains instructions has the same effect with respect to the sequential execution model as a Store instruction as described there.)

For the instructions and facilities defined in this Book, there are two additional exceptions to the sequential execution model that the processor obeys beyond those described in Section 2.2 of Book I.

- an event-based branch occurs (see Chapter 6)
- the BHRB is read (see Section 7.2)

- event-based exception
  An unusual condition, or external signal, that sets a status bit in the BESCR and may or may not cause an event-based branch, depending upon whether event-based branches are enabled.

- storage location
  A contiguous sequence of one or more bytes in storage. When used in association with a specific instruction or the instruction fetching mechanism, the length of the sequence of one or more bytes is typically implied by the operation. In other uses, it may refer more abstractly to a group of bytes which share common storage attributes.

- storage access
  An access to a storage location. There are three (mutually exclusive) kinds of storage access.

  - data access
    An access to the storage location specified by a Load or Store instruction, or, if the access is
performed “out-of-order” (see Section 6.5 of Book III), an access to a storage location as if it were the storage location specified by a Load or Store instruction.

- instruction fetch
  An access for the purpose of fetching an instruction.

- implicit access
  An access by the processor for the purpose of finding the address translation tables, translating an address, or recording reference and change information (see Book III).

- caused by, associated with
  - caused by
    A storage access is said to be caused by an instruction if the instruction is a Load or Store and the access (data access) is to the storage location specified by the instruction.
  - associated with
    A storage access is said to be associated with an instruction if the access is for the purpose of fetching the instruction (instruction fetch), or is a data access caused by the instruction, or is an implicit access that occurs as a side effect of fetching or executing the instruction.

- metadata
  Companion data associated with a storage location. In addition to the data that is loaded into a target register or stored from a source register, a storage location may be associated with additional control information. The granularity, meaning, and treatment of the control information may vary based on the type of storage access involved and on the state of the process making the access. Unless otherwise stated or obvious from context, loads ignore the metadata and stores zero the metadata.

Note that not all storage locations have associated metadata. The absence of associated metadata does not necessarily prevent successful completion of an instruction that specifies the treatment of metadata. Unless otherwise stated or obvious from context, metadata associated with a store is lost, and a load will have its metadata set to zero if the storage location has no associated metadata.

- prefetched instructions
  Instructions for which a copy of the instruction has been fetched from instruction storage, but the instruction has not yet been executed.

- uniprocessor
  A system that contains one processor.

- multiprocessor
  A system that contains two or more processors.

- shared storage multiprocessor
  A multiprocessor that contains some common storage, which all the processors in the system can access.

- performed
  A load or instruction fetch by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is performed with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently). An instruction cache block invalidation by P1 is performed with respect to P2 when the instruction that requested the invalidation has caused the specified block, if present, to be made invalid in P2’s instruction cache, and similarly for a data cache block invalidation.

The preceding definitions apply regardless of whether P1 and P2 are the same entity.

- page (virtual page)
  2<sup>n</sup> contiguous bytes of storage aligned such that the effective address of the first byte in the page is an integral multiple of the page size for which protection and control attributes are independently specifiable and for which reference and change status are independently recorded.

- block
  The aligned unit of storage operated on by the Cache Management instructions. The size of an instruction cache block may differ from the size of a data cache block, and both sizes may vary between implementations. The maximum block size is equal to the minimum page size.

### 1.2 Introduction

The Power ISA User Instruction Set Architecture, discussed in Book I, defines storage as a linear array of bytes indexed from 0 to a maximum of 2<sup>64</sup>-1. Each byte is identified by its index, called its address, and each byte contains a value. This information is sufficient to allow the programming of applications that require no special features of any particular system environment. The Power ISA Virtual Environment Architecture, described herein, expands this simple storage model to include caches, virtual storage, and shared storage multiprocessors. The Power ISA Virtual Environment Architecture, in conjunction with services based on the Power ISA Operating Environment Architecture (see Book III) and provided by the operating system, permits explicit control of this expanded storage model. A sim-
ple model for sequential execution allows at most one storage access to be performed at a time and requires that all storage accesses appear to be performed in program order. In contrast to this simple model, the Power ISA specifies a relaxed model of storage consistency. In a multiprocessor system that allows multiple copies of a storage location, aggressive implementations of the architecture can permit intervals of time during which different copies of a storage location have different values. This chapter describes features of the Power ISA that enable programmers to write correct programs for this storage model.

1.3 Virtual Storage

The Power ISA system implements a virtual storage model for applications. This means that a combination of hardware and software can present a storage model that allows applications to exist within a “virtual” address space larger than either the effective address space or the real address space.

Each program can access 2^{64} bytes of “effective address” (EA) space, subject to limitations imposed by the operating system. In a typical Power ISA system, each program’s EA space is a subset of a larger “virtual address” (VA) space managed by the operating system.

Each effective address is translated to a real address (i.e., to an address of a byte in real storage or on an I/O device) before being used to access storage. The hardware accomplishes this, using the address translation mechanism described in Book III. The operating system manages the real (physical) storage resources of the system, by setting up the tables and other information used by the hardware address translation mechanism.

In general, real storage may not be large enough to map all the virtual pages used by the currently active applications. With support provided by hardware, the operating system can attempt to use the available real pages to map a sufficient set of virtual pages of the applications. If a sufficient set is maintained, “paging” activity is minimized. If not, performance degradation is likely.

The operating system can support restricted access to virtual pages (including read/write, read only, and no access; see Book III), based on system standards (e.g., program code might be read only) and application requests.

1.4 Single-Copy Atomicity

An access is single-copy atomic, or simply atomic, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus serialized: each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors.

The access caused by an instruction other than a Load/Store Multiple or Move Assist instruction is guaranteed to be atomic if the storage operand is not larger than a doubleword and is aligned (see Section 1.10.1 of Book I).

Quadword accesses with aligned storage operands are guaranteed to be atomic when caused by the following instructions:
- `lq`
- `stq`
- `lqarx`
- `stqcx`.

Quadword atomicity applies only to storage that is neither Write Through Required nor Caching Inhibited. The cases described above are the only cases in which the access to the storage operand is guaranteed to be atomic. For example, the access caused by the following instructions is not guaranteed to be atomic:
- any Load or Store instruction for which the storage operand is unaligned
- `lmw, stmw, lswi, lswx, stswi, stswx`
- `fldp, fldpx, stfdp, stfdpx`
- any Cache Management instruction

An access that is not atomic is performed as a set of smaller disjoint atomic accesses. If the non-atomic access is caused by an instruction other than a Load/Store Multiple or Move Assist instruction and one of the following conditions is satisfied, the non-atomic access is performed as described in the corresponding list item. The first list item matching a given situation applies.

- The storage operand is at least 16 bytes long and is doubleword-aligned:
  - the access is performed as a set of disjoint atomic accesses each of which consists of one or more aligned doublewords.
- The storage operand is at least eight bytes long and is word-aligned:
  - the access is performed as a set of disjoint atomic accesses each of which consists of one or more aligned words.
- The storage operand is at least four bytes long and is halfword-aligned:
  - the access is performed as a set of disjoint atomic accesses each of which consists of one or more aligned halfwords.

In all other cases the number, length, and alignment of the component disjoint atomic accesses are implementation-dependent. In all cases the relative order in which the component disjoint atomic accesses are performed is implementation-dependent.

The results for several combinations of loads and stores to the same or overlapping locations are described below.
1. When two processors perform atomic stores to locations that do not overlap, and no other stores are performed to those locations, the contents of those locations are the same as if the two stores were performed by a single processor.

2. When two processors perform atomic stores to the same storage location, and no other store is performed to that location, the contents of that location are the result stored by one of the processors.

3. When two processors perform stores that have the same target location and are not guaranteed to be atomic, and no other store is performed to that location, the result is some combination of the bytes stored by both processors.

4. When two processors perform stores to overlapping locations, and no other store is performed to those locations, the result is some combination of the bytes stored by the processors to the overlapping bytes. The portions of the locations that do not overlap contain the bytes stored by the processor storing to the location.

5. When a processor performs an atomic store to a location, a second processor performs an atomic load from that location, and no other store is performed to that location, the value returned by the load is the contents of the location before the store or the contents of the location after the store.

6. When a load and a store with the same target location can be performed simultaneously, and the accesses are not guaranteed to be atomic, and no other store is performed to that location, the value returned by the load is some combination of the contents of the location before the store and the contents of the location after the store.

1.5 Cache Model

A cache model in which there is one cache for instructions and another cache for data is called a “Harvard-style” cache. This is the model assumed by the Power ISA, e.g., in the descriptions of the Cache Management instructions in Section 4.3. Alternative cache models may be implemented (e.g., a “combined cache” model, in which a single cache is used for both instructions and data, or a model in which there are several levels of caches), but they support the programming model implied by a Harvard-style cache.

The processor is not required to maintain copies of storage locations in the instruction cache consistent with modifications to those storage locations (e.g., modifications caused by Store instructions).

A location in the data cache is considered to be modified in that cache if the location has been modified (e.g., by a Store instruction) and the modified data have not been written to main storage.

Cache Management instructions are provided so that programs can manage the caches when needed. For example, program management of the caches is needed when a program generates or modifies code that will be executed (i.e., when the program modifies data in storage and then attempts to execute the modified data as instructions). The Cache Management instructions are also useful in optimizing the use of memory bandwidth in such applications as graphics and numerically intensive computing. The functions performed by these instructions depend on the storage control attributes associated with the specified storage location (see Section 1.6, “Storage Control Attributes”).

The Cache Management instructions allow the program to do the following.

- invalidate the copy of storage in an instruction cache block (icbl)
- provide a hint that an instruction will probably soon be accessed from a specified instruction cache block (icbt)
- provide a hint that the program will probably soon access a specified data cache block (dcbt, dcbst)
- set the contents of a data cache block to zeros (dcbz)
- copy the contents of a modified data cache block to main storage (dcbst)
- copy the contents of a modified data cache block to main storage and make the copy of the block in the data cache invalid (dcbf or dcblf)

A write to main storage caused by a dcbst or dcbf instruction has updated main storage when a load by any given processor, that is satisfied from main storage from the location accessed by the write, will return the value written (or a value written or stored subsequently), and similarly for the store caused by a Store instruction.

1.6 Storage Control Attributes

Some operating systems may provide a means to allow programs to specify the storage control attributes described in this section. Because the support provided for these attributes by the operating system may vary between systems, the details of the specific system being used must be known before these attributes can be used.

Storage control attributes are associated with units of storage that are multiples of the page size. Each storage access is performed according to the storage control attributes of the specified storage location, as described below. The storage control attributes are the following.

- Write Through Required
- Caching Inhibited
Memory Coherence Required
Guarded

These attributes have meaning only when an effective address is translated by the processor performing the storage access.

Programming Note

The Write Through Required and Caching Inhibited attributes are mutually exclusive because, as described below, the Write Through Required attribute permits the storage location to be in the data cache while the Caching Inhibited attribute does not.

Storage that is Write Through Required or Caching Inhibited is not intended to be used for general-purpose programming. For example, the lbarx, llbarx, lwarx, ldarx, lg0rx, shtbcx, sthcx, stwbcx, stdcx, and stqcx instructions may cause the system data storage error handler to be invoked if they specify a location in storage having either of these attributes. To obtain the best performance across the widest range of implementations, storage that is Write Through Required or Caching Inhibited should be used only when the use of such storage meets specific functional or semantic needs or enables a performance optimization.

In the remainder of this section, "Load instruction" includes the Cache Management and other instructions that are stated in the instruction descriptions to be "treated as a Load" unless they are explicitly excluded, and similarly for "Store instruction".

1.6.1 Write Through Required

A store to a Write Through Required storage location is performed in main storage. A Store instruction that specifies a location in Write Through Required storage may cause additional locations in main storage to be accessed. If a copy of the block containing the specified location is retained in the data cache, the store is also performed in the data cache. The store does not cause the block to be considered to be modified in the data cache.

In general, accesses caused by separate Store instructions that specify locations in Write Through Required storage may be combined into one access. Such combining does not occur if the Store instructions are separated by a sync or eieio instruction.

1.6.2 Caching Inhibited

An access to a Caching Inhibited storage location is performed in main storage. A Load instruction that specifies a location in Caching Inhibited storage may cause additional locations in main storage to be accessed unless the specified location is also Guarded. An instruction fetch from Caching Inhibited storage may cause additional words in main storage to be accessed. No copy of the accessed locations is placed into the caches.

In general, non-overlapping accesses caused by separate Load instructions that specify locations in Caching Inhibited storage may be combined into one access, as may non-overlapping accesses caused by separate Store instructions that specify locations in Caching Inhibited storage. Such combining does not occur if the Load or Store instructions are separated by a sync instruction. Combining may also occur among such accesses from multiple processors that share a common memory interface. No combining occurs if the storage is also Guarded.

Programming Note

None of the memory barrier instructions prevent the combining of accesses from different processors. The Guarded storage attribute must be used in combination with Caching Inhibited to prevent such combining.

1.6.3 Memory Coherence Required

An access to a Memory Coherence Required storage location is performed coherently, as follows.

Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are coherent if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the physical storage location need not assume each of the values written to it. For example, a processor may update a location several times before the value is written to physical storage. The result of a store operation is not available to every processor or mechanism at the same instant, and it may be that a processor or mechanism observes only some of the values that are written to a location. However, when a location is accessed atomically and coherently by all processors and mechanisms, the sequence of values loaded from the location by any processor or mechanism during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor or mechanism can never load a “newer” value first and then, later, load an “older” value.

Memory coherence is managed in blocks called coherence blocks. Their size is implementation-dependent, but is larger than a word and is usually the size of a cache block.

For storage that is not Memory Coherence Required, software must explicitly manage memory coherence to...
the extent required by program correctness. The operations required to do this may be system-dependent.

Because the Memory Coherence Required attribute for a given storage location is of little use unless all processors that access the location do so coherently, in statements about Memory Coherence Required storage elsewhere in this document it is generally assumed that the storage has the Memory Coherence Required attribute for all processors that access it.

### Programming Note

Operating systems that allow programs to request that storage not be Memory Coherence Required should provide services to assist in managing memory coherence for such storage, including all system-dependent aspects thereof.

In most systems the default is that all storage is Memory Coherence Required. For some applications in some systems, software management of coherence may yield better performance. In such cases, a program can request that a given unit of storage not be Memory Coherence Required, and can manage the coherence of that storage by using the `sync` instruction, the `Cache Management` instructions, and services provided by the operating system.

1.6.4 Guarded

A data access to a Guarded storage location is performed only if either (a) the access is caused by an instruction that is known to be required by the sequential execution model, or (b) the access is a load and the storage location is already in a cache. If the storage is also Caching Inhibited, only the storage location specified by the instruction is accessed; otherwise any storage location in the cache block containing the specified storage location may be accessed.

Except in ultravisor or hypervisor real addressing mode, instructions are not fetched from storage that is Guarded. Except in these addressing modes, if the instruction addressed by the current instruction address is in such storage, the system instruction storage error handler is invoked (see Section 7.5.5 of Book III).

---

1.7 Shared Storage

This architecture supports the sharing of storage between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to a storage location by one or more programs using different effective addresses. All these cases are considered storage sharing. Storage is shared in blocks that are an integral number of pages.

When the same storage location has different effective addresses, the addresses are said to be aliases. Each application can be granted separate access privileges to aliased pages.

1.7.1 Storage Access Ordering

The storage model for the ordering of storage accesses is weakly consistent. This model provides an opportunity for improved performance over a model that has stronger consistency rules, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed when storage is shared by two or more programs.

The order in which the processor performs storage accesses, the order in which those accesses are performed with respect to another processor or mechanism, and the order in which those accesses are performed in main storage may all be different. Several means of enforcing an ordering of storage accesses are provided to allow programs to share storage with other programs, or with mechanisms such as I/O devices. These means are listed below. The phrase "to the extent required by the associated Memory
Coherence Required attributes” refers to the Memory Coherence Required attribute, if any, associated with each access.

- If two Store instructions or two Load instructions specify storage locations that are both Caching Inhibited and Guarded, the corresponding storage accesses are performed in program order with respect to any processor or mechanism.

- If a Load instruction depends on the value returned by a preceding Load instruction (because the value is used to compute the effective address specified by the second Load), the corresponding storage accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated Memory Coherence Required attributes. This applies even if the dependency has no effect on program logic (e.g., the value returned by the first Load is ANDed with zero and then added to the effective address specified by the second Load).

- When a processor (P1) executes a Synchronize or eieio instruction a memory barrier is created, which orders applicable storage accesses pairwise, as follows. Let A be a set of storage accesses that includes all storage accesses associated with instructions preceding the barrier-creating instruction, and let B be a set of storage accesses that includes all storage accesses associated with instructions following the barrier-creating instruction. For each applicable pair $a_i,b_j$ of storage accesses such that $a_i$ is in A and $b_j$ is in B, the memory barrier ensures that $a_i$ will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before $b_j$ is performed with respect to that processor or mechanism.

The ordering done by a memory barrier is said to be “cumulative” if it also orders storage accesses that are performed by processors and mechanisms other than P1, as follows.

- A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.

- B includes all applicable storage accesses by any such processor or mechanism that are performed after a Load instruction executed by that processor or mechanism has returned the value stored by a store that is in B.

No ordering should be assumed among the storage accesses caused by a single instruction (i.e., by an instruction for which the access is not atomic), and no means are provided for controlling that order.
Programming Note

Because stores cannot be performed “out-of-order” (see Book III), if a Store instruction depends on the value returned by a preceding Load instruction (because the value returned by the Load is used to compute either the effective address specified by the Store or the value to be stored), the corresponding storage accesses are performed in program order. The same applies if whether the Store instruction is executed depends on a conditional Branch instruction that in turn depends on the value returned by a preceding Load instruction.

Because an isync instruction prevents the execution of instructions following the isync until instructions preceding the isync have completed, if an isync follows a conditional Branch instruction that depends on the value returned by a preceding Load instruction, the load on which the Branch depends is performed before any loads caused by instructions following the isync. This applies even if the effects of the “dependency” are independent of the value loaded (e.g., the value is compared to itself and the Branch tests the EQ bit in the selected CR field), and even if the branch target is the sequentially next instruction.

With the exception of the cases described above and earlier in this section, data dependencies and control dependencies do not order storage accesses. Examples include the following.

- If a Load instruction specifies the same storage location as a preceding Store instruction and the location is in storage that is not Caching Inhibited, the load may be satisfied from a “store queue” (a buffer into which the processor places stored values before presenting them to the storage subsystem), and not be visible to other processors and mechanisms. A consequence is that if a subsequent Store depends on the value returned by the Load, the two stores need not be performed in program order with respect to other processors and mechanisms.

- Because a Store Conditional instruction may complete before its store has been performed, a conditional Branch instruction that depends on the CR0 value set by a Store Conditional instruction does not order the Store Conditional’s store with respect to storage accesses caused by instructions that follow the Branch.

- Because processors may predict branch target addresses and branch condition resolution, control dependencies (e.g., branches) do not order storage accesses except as described above. For example, when a subroutine returns to its caller the return address may be predicted, with the result that loads caused by instructions at or after the return address may be performed before the load that obtains the return address is performed.

Because processors may implement nonarchitected duplicates of architected resources (e.g., GPRs, CR fields, and the Link Register), resource dependencies (e.g., specification of the same target register for two Load instructions) do not order storage accesses.

Examples of correct uses of dependencies, sync and lwsync to order storage accesses can be found in Appendix B. “Programming Examples for Sharing Storage” on page 1107.

Because the storage model is weakly consistent, the sequential execution model as applied to instructions that cause storage accesses guarantees only that those accesses appear to be performed in program order with respect to the processor executing the instructions. For example, an instruction may complete, and subsequent instructions may be executed, before storage accesses caused by the first instruction have been performed. However, for a sequence of atomic accesses to the same storage location, if the location is in storage that is Memory Coherence Required the definition of coherence guarantees that the accesses are performed in program order with respect to any processor or mechanism that accesses the location coherently, and similarly if the location is in storage that is Caching Inhibited.

Because accesses to storage that is Caching Inhibited are performed in main storage, memory barriers and dependencies on Load instructions order such accesses with respect to any processor or mechanism even if the storage is not Memory Coherence Required.
1.7.1.1 Storage Ordering of Copy/Paste-Initiated Data Transfers

The Copy-Paste Facility (see Section 4.4) uses pairs of instructions to initiate 128-byte data transfers. They are referred to as "data transfers" to differentiate them from the "normal" storage accesses caused by or associated with loads, stores, and instructions that are treated as loads and stores. In the absence of barriers, the relative ordering among adjacent data transfers or data transfers and storage accesses is not defined, and the sequential execution model and coherence-required ordering relationships do not apply. To establish order between adjacent data transfers or between data transfers and storage accesses, `hwsync` must be used. See the description of the `Synchronize` instruction in Section 4.6.3 for more information.

1.7.1.2 Storage Ordering of Stores to Persistent Storage

A location in a region of main storage that is backed by persistent storage is considered to be modified relative to persistent storage if the location has been modified in main storage (e.g., by a Store instruction) and the modified data have not been written to persistent storage. A store has updated persistent storage when a load by any given processor, from the location accessed by the store, would return the value stored (or a value stored subsequently) if system power were lost temporarily between the time the store has updated persistent storage and the time the load is performed. A store may update persistent storage significantly later than it updates main storage. The `dcbstps` (data cache block store to persistent storage) and `dcbffsp` (data cache block flush to persistent storage) instructions can be used to write modified locations in a block to persistent storage (and to perform the functions of Data Cache Block Store and Data Cache Block Flush respectively). The `phwsync` (persistent heavy-weight sync) and `plwsync` (persistent lightweight sync) instructions can be used to establish order for these writes to persistent storage. A store may update persistent storage even in the absence of `dcbstps` and `dcbffps` instructions targeting the cache block(s) affected by the store. See Section 4.3.2 and Section 4.6.3 for more information.

Except in this section and in other material dealing with persistent storage, references to the storage subsystem in this document assume system power is not lost unless otherwise stated or obvious from context.

Programming Note

It may be helpful to think of a `copy/paste`, pair sending the real storage addresses of the 128-byte source and destination to an asynchronous data transfer engine completely separate from the processor that is executing the `copy` and `paste` instructions. The data transfers collect in the engine's queue. The engine may perform the data transfers in any order, and with the only relative timing relationship to adjacent transfers and accesses being determined by `hwsync`.

Programming Note

The first example below illustrates cumulative ordering of storage accesses preceding a memory barrier, and the second illustrates cumulative ordering of storage accesses following a memory barrier. Assume that locations X, Y, and Z initially contain the value 0.

Example 1:
Processor A:
stores the value 1 to location X

Processor B:
loads from location X obtaining the value 1, executes a `sync` instruction, then stores the value 2 to location Y

Processor C:
loads from location Y obtaining the value 2, executes a `sync` instruction, then loads from location X

Example 2:
Processor A:
stores the value 1 to location X, executes a `sync` instruction, then stores the value 2 to location Y

Processor B:
loops loading from location Y until the value 2 is obtained, then stores the value 3 to location Z

Processor C:
loads from location Z obtaining the value 3, executes a `sync` instruction, then loads from location X

In both cases, cumulative ordering dictates that the value loaded from location X by processor C is 1.
1.7.1.3 Storage Ordering of I/O Accesses

A “coherence domain” consists of all processors and all interfaces to main storage. Memory reads and writes initiated by mechanisms outside the coherence domain are performed within the coherence domain in the order in which they enter the coherence domain and are performed as coherent accesses.

1.7.2 Atomic Update

The Load And Reserve and Store Conditional instructions together permit atomic update of a shared storage location. There are byte, halfword, word, doubleword, and quadword forms of each of these instructions. Described here is the operation of the word forms \textit{lwarx} and \textit{stwcx}; operation of the byte, halfword, doubleword, and quadword forms \textit{lbax}, \textit{stbx}, \textit{lwax}, \textit{stdcx}, \textit{ldcx}, \textit{ldcx}, and \textit{stqcx} is the same except for obvious substitutions.

The \textit{lwarx} instruction is a load from a word-aligned location that has two side effects. Both of these side effects occur at the same time that the load is performed.

1. A reservation for a subsequent \textit{stwcx} instruction is created.

2. The memory coherence mechanism is notified that a reservation exists for the storage location specified by the \textit{lwarx}.

The \textit{stwcx} instruction is a store to a word-aligned location that is conditioned on the existence of the reservation created by the \textit{lwarx} and on whether the same storage location is specified by both instructions. To emulate an atomic operation with these instructions, it is necessary that both the \textit{lwarx} and the \textit{stwcx} specify the same storage location.

A \textit{stwcx} performs a store to the target storage location only if the reservation created by the \textit{lwarx} still exists at the time the \textit{stwcx} is executed, and only if the storage locations specified by the two instructions are in the same aligned block of real storage whose size is the smallest real page size supported by the implementation. The remainder of this paragraph assumes that these two conditions are satisfied. If the storage locations specified by the two instructions differ, or if a \textit{Store Conditional} instruction is used with a preceding \textit{Load And Reserve} instruction that has a different storage operand length (e.g., \textit{stwcx} with \textit{ldarx}), whether the store is performed is undefined. Otherwise the store is performed.

A \textit{stwcx} that performs its store is said to “succeed”.

Examples of the use of \textit{lwarx} and \textit{stwcx} are given in Appendix B. “Programming Examples for Sharing Storage” on page 1107.

A successful \textit{stwcx} to a given location may complete before its store has been performed with respect to other processors and mechanisms. As a result, a subsequent load or \textit{lwarx} from the given location by another processor may return a “stale” value. However, a subsequent \textit{lwarx} from the given location by the other processor followed by a successful \textit{stwcx} by that processor is guaranteed to have returned the value stored by the first processor’s \textit{stwcx}. (In the absence of other stores to the given location).
**Programming Note**

If a virtual address is reassigned to a different real page, a reservation established at the virtual address before the reassignment will not be cleared by a store to the new real page by some other processor or mechanism. (As described in Section 1.7.2.1, reservations are held on real addresses.) If **Store Conditional** instructions did not suppress the store when the storage location specified by the **Store Conditional** instruction is in a different real page from the storage location specified by the corresponding **Load And Reserve** instruction, such virtual address reassignment could permit a **Store Conditional** instruction that specifies the same virtual address as the corresponding **Load And Reserve** instruction, and logically should fail because the other processor or mechanism stored to the virtual address, to succeed.

This real address checking cannot detect that the virtual page in which the reservation was established has been moved to a new real page and back again to the original real page that was accessed by the **Load And Reserve** instruction. It also cannot detect that the real address of the storage location specified by a **Store Conditional** instruction is the same as the real address of the reservation, or is in the same real page as the reservation, only because the virtual page containing the storage location specified by the **Store Conditional** instruction has been moved to the real page that was accessed by the corresponding **Load And Reserve** instruction. Privileged software that moves a virtual page should clear the reservation on the processor it is running on in order to ensure that a **Store Conditional** instruction executed by that processor does not succeed in these cases. (If the software that moves the virtual page uses **Load And Reserve** and **Store Conditional** for its own purposes, the clearing of the original reservation will happen naturally. The stores that occur naturally as part of moving the virtual page will cause any reservations, held by other processors, in the target real page to be cleared.)

**1.7.2.1 Reservations**

The ability to emulate an atomic operation using **lwarx** and **stwcx**, is based on the conditional behavior of **stwcx**, the reservation created by **lwarx**, and the clearing of that reservation if the target storage location is modified by another processor or mechanism before the **stwcx** performs its store.

A reservation is held on an aligned unit of real storage called a reservation granule. The size of the reservation granule is \(2^n\) bytes, where \(n\) is implementation-dependent but is always at least 4 (thus the minimum reservation granule size is a quadword), and where \(2^n\) is not larger than the smallest real page size supported by the implementation. The reservation granule associated with effective address EA contains the real address to which EA maps. ("real_addr(EA)" in the RTL for the **Load And Reserve** and **Store Conditional** instructions stands for "real address to which EA maps"). The reservation also has an associated length, which is equal to the storage operand length, in bytes, of the **Load And Reserve** instruction that established the reservation.

A processor has at most one reservation at any time. A reservation is established by executing a **lbarx**, **lharx**, **lwarx**, **ldarx**, or **ldarx** instruction, as described in item 1 below, and is lost or may be lost, depending on the item, if any of the following occur. Items 1-8 apply only if the relevant access is performed. (For example, an access that would ordinarily be caused by an instruction might not be performed if the instruction causes the system error handler to be invoked.)

1. The processor holding the reservation executes another **lbarx**, **lbarx**, **lwarx**, **ldarx**, or **ldarx**: this clears the first reservation and establishes a new one.

2. The processor holding the reservation executes any **stbcx**, **sthcx**, **stwcx**, **stdcx**, or **stqcx**, regardless of whether the specified address matches the address specified by the **lbarx**, **lbarx**, **lwarx**, **ldarx**, or **ldarx** that established the reservation, and regardless of whether the storage operands lengths of the two instructions are the same.

3. The processor holding the reservation executes an **AMO** that updates the same reservation granule: whether the reservation is lost is undefined.

4. Some other processor executes a **Store or dcbz** that specifies a location in the same reservation granule.

5. Some other processor executes a **dcbstx**, or **dcbst** that specifies a location in the same reservation granule: whether the reservation is lost is undefined. (For a **dcbstx** instruction that specifies a data stream, "location" in the preceding sentence includes all locations in the data stream.)

6. Any processor modifies a Reference or Change bit in the same reservation granule: the reservation is lost if the modification is atomic; otherwise whether the reservation is lost is undefined. (See Section 6.7.12 of Book III)

7. Some mechanism other than a processor modifies a storage location in the same reservation granule.

8. An interrupt (see Book III) occurs on the processor holding the reservation: the interrupt itself does not clear the reservation, but system software invoked by the interrupt may clear the reservation.

9. Implementation-specific characteristics of the coherence mechanism cause the reservation to be lost.
The reservation is also used by the `waitrsv` instruction (see Section 4.6.4).

**Virtualized Implementation Note**

A reservation may be lost if:
- Software executes a privileged instruction or utilizes a privileged facility
- Software accesses storage not intended for general-purpose programming

**Programming Note**

One use of `lwarx` and `stwcx` is to emulate a “Compare and Swap” primitive like that provided by the IBM System/370 Compare and Swap instruction; see Section B.1, “Atomic Update Primitives” on page 1107. A System/370-style Compare and Swap checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The combination of `lwarx` and `stwcx` improves on such a Compare and Swap, because the reservation reliably binds the `lwarx` and `stwcx` together. The reservation is always lost if the word is modified by another processor or mechanism between the `lwarx` and `stwcx`, so the `stwcx` never succeeds unless the word has not been stored into (by another processor or mechanism) since the `lwarx`.

**Programming Note**

In general, programming conventions must ensure that `lwarx` and `stwcx` specify addresses that match; a `stwcx` should be paired with a specific `lwarx` to the same storage location. Situations in which a `stwcx` may erroneously be issued after some `lwarx` other than that with which it is intended to be paired must be scrupulously avoided. For example, there must not be a context switch in which the processor holds a reservation in behalf of the old context, and the new context resumes after a `lwarx` and before the paired `stwcx`. The `stwcx` in the new context might succeed, which is not what was intended by the programmer. Such a situation must be prevented by executing a `stbcx`, `stthcx`, `stwcx`, `stdcx`, or `stqcx` that specifies a dummy writable aligned location as part of the context switch; see Section 7.4.3 of Book III.

**Programming Note**

Because the reservation is lost if another processor stores anywhere in the reservation granule, lock words (or bytes, halfwords, or doublewords) should be allocated such that few such stores occur, other than perhaps to the lock word itself. (Stores by other processors to the lock word result from contention for the lock, and are an expected consequence of using locks to control access to shared storage: stores to other locations in the reservation granule can cause needless reservation loss.) Such allocation can most easily be accomplished by allocating an entire reservation granule for the lock and wasting all but one word. Because reservation granule size is implementation-dependent, portable code must do such allocation dynamically.

Similar considerations apply to other data that are shared directly using `lwarx` and `stwcx` (e.g., pointers in certain linked lists; see Section B.3, “List Insertion” on page 1111).

### 1.7.2.2 Forward Progress

Forward progress in loops that use `lwarx` and `stwcx` is achieved by a cooperative effort among hardware, system software, and application software.

The architecture guarantees that when a processor executes an `lwarx` to obtain a reservation for location X and then an `stwcx` to store a value to location X, either

1. the `stwcx` succeeds and the value is written to location X, or
2. the `stwcx` fails because some other processor or mechanism modified location X, or
3. the `stwcx` fails because the processor’s reservation was lost for some other reason.

In Cases 1 and 2, the system as a whole makes progress in the sense that some processor successfully modifies location X. Case 3 covers reservation loss required for correct operation of the rest of the system. This includes cancellation caused by some other processor or mechanism writing elsewhere in the reservation granule, cancellation caused by the operating system in managing certain limited resources such as real storage, and cancellation caused by any of the other effects listed in see Section 1.7.2.1.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in Case 3. While the architecture alone cannot provide such a guarantee, the characteristics listed in Cases 1 and 2 are necessary conditions for any forward progress
guarantee. An implementation and operating system can build on them to provide such a guarantee.

---

**Virtualized Implementation Note**

On a virtualized implementation, Case 3 includes reservation loss caused by the virtualization software. Thus, on a virtualized implementation, a reservation may be lost at any time without apparent cause. The virtualization software participates in any forward progress assurances, as described above.

---

**Programming Note**

The architecture does not include a “fairness guarantee”. In competing for a reservation, two processors can indefinitely lock out a third.

---

### 1.8 Instruction Storage

The instruction execution properties and requirements described in this section, including its subsections, apply only to instruction execution that is required by the sequential execution model.

In this section, including its subsections, it is assumed that all instructions for which execution is attempted are in storage that is not Caching Inhibited and (unless instruction address translation is disabled; see Book III) is not Guarded, and from which instruction fetching does not cause the system error handler to be invoked (e.g., from which instruction fetching is not prohibited by the "address translation mechanism" or the "storage protection mechanism"; see Book III).

---

**Programming Note**

The results of attempting to execute instructions from storage that does not satisfy this assumption are described in Section 1.6.2 and Section 1.6.4 of this Book and in Book III.

---

The instruction cache is not necessarily kept consistent with the data cache or with main storage. It is the responsibility of software to ensure that instruction storage is consistent with data storage when such consistency is required for program correctness.

After one or more bytes of a storage location have been modified and before an instruction located in that storage location is executed, software must execute the appropriate sequence of instructions to make instruction storage consistent with data storage. Otherwise the result of attempting to execute the instruction is boundedly undefined except as described in Section 1.8.1, "Concurrent Modification and Execution of Instructions" on page 1043.
Programming Note

Following are examples of how to make instruction storage consistent with data storage. Because the optimal instruction sequence to make instruction storage consistent with data storage may vary between systems, many operating systems will provide a system service to perform this function.

Case 1: The given program does not modify instructions executed by another program nor does another program modify the instructions executed by the given program.

Assume that location X previously contained the instruction A0; the program modified one or more bytes of that location such that, in data storage, the location contains the instruction A1; and location X is wholly contained in a single cache block. The following instruction sequence will make instruction storage consistent with data storage such that if the isync was in location X-4, the instruction A1 in location X would be executed immediately after the isync.

```
dcbst X #copy the block in main storage
csync #order copy before invalidation
icbi X #invalidate copy in instr cache
isync #order invalidation before store
    # to flag
stw r0,flag #set flag indicating instruction
    # storage is now consistent
```

Case 2: One or more programs execute the instructions that are concurrently being modified by another program.

Assume program A has modified the instruction at location X and other programs are waiting for program A to signal that the new instruction is ready to execute. The following instruction sequence will make instruction storage consistent with data storage and then set a flag to indicate to the waiting programs that the new instruction can be executed.

```
li r0,1 #put a 1 value in r0
dcbst X #copy the block in main storage
csync #order copy before invalidation
icbi X #invalidate copy in instr cache
csync #order invalidation before store
    # to flag
stw r0,flag #set flag indicating instruction
    # storage is now consistent
```

The following instruction sequence, executed by the waiting program, will prevent the waiting programs from executing the instruction at location X until location X in instruction storage is consistent with data storage, and then will cause any prefetched instructions to be discarded.

```
lwz r0,flag #loop until flag = 1 (when 1 is loaded, location X in inst'n
    be stored is consistent with
    # location X in data storage)
    # to flag
icbi X #invalidate copy in instr cache
icbi X #invalidate copy in instr cache
csync #order invalidation before store
    # to flag
```

In the preceding instruction sequence any context synchronizing instruction (e.g., rfid) can be used instead of isync. (For Case 1 only isync can be used.)

For both cases, if two or more instructions in separate data cache blocks have been modified, the dcbst instruction in the examples must be replaced by a sequence of dcbst instructions such that each block containing the modified instructions is copied back to main storage. Similarly, for icbi the sequence must invalidate each instruction cache block containing a location of an instruction that was modified. The sync instruction that appears above between “dcbst X” and “icbi X” would be placed between the sequence of dcbst instructions and the sequence of icbi instructions.
1.8.1 Concurrent Modification and Execution of Instructions

The phrase “concurrent modification and execution of instructions” (CMODX) refers to the case in which a processor fetches and executes an instruction from instruction storage which has not been made consistent with data storage. This section describes the only case in which executing this instruction under these conditions produces defined results.

In the remainder of this section the following terminology is used.

- Location X is an arbitrary four-byte word-aligned storage location.
- \( X_0 \) is the value of the contents of location X for which software has made the location X in instruction storage consistent with data storage.
- \( X_1, X_2, \ldots, X_n \) are the sequence of the first n values occupying location X after \( X_0 \).
- \( X_n \) is the first value of X subsequent to \( X_0 \) for which software has again made instruction storage consistent with data storage.

- The “patch class” of words consists of the following.
  - the I-form Branch instruction \((b[l][a])\)
  - the preferred no-op instruction \((ori 0,0,0)\)
  - the prefix of the Prefixed No-op instruction \((pnop)\)
  - the D-form and X-form Trap instructions for which \( TO = 0b11111 \)

- The “instruction from location X” includes both the case of a word instruction contained in location X and the case of a prefixed instruction for which the prefix is contained in location X.

If the instruction from location X is executed after the copy of location X in instruction storage is made consistent for the value \( X_0 \) and before it is made consistent for the value \( X_n \), the results of executing the instruction are defined if and only if the following conditions are satisfied.

1. The stores that place the values \( X_1, \ldots, X_n \) into location X are atomic stores that modify all four bytes of location X.
2. The sequence of \( X_i \) values is one of two types:
   - any sequence in which each \( X_i \) is a patch class word, or
   - any sequence that is comprised of exactly two unique word values, at least one of which is a patch class word.
3. If a sequence of \( X_i \) values contains a prefix, the only word instructions the sequence can contain are the patch class Branch and patch class Trap instructions. In this use, the target of the Branch instruction must not be the word following the Branch instruction.
4. Location X is in storage that is Memory Coherence Required.

Programming Note

The architecture does not support CMODX modification of an entire prefixed instruction because it may be unaligned and therefore impossible to modify atomically. Furthermore, no need has arisen to motivate the creation of special cases where it must work. The architecture also does not support CMODX modification of the suffix of a prefixed instruction.

If these conditions are satisfied, each execution of an instruction from location X will use some \( X_i \), \( 0 \leq i \leq n \). The value of the ordinate \( i \) associated with each value executed may be different and the sequence of ordinates \( i \) associated with a sequence of values executed is not constrained, (e.g., a valid sequence of executions of the instruction from location X could use the sequence \( X_i, X_{i+2}, \) then \( X_{i+1} \)). If these conditions are not satisfied, the results of each such execution of an instruction from location X are boundedly undefined, and may include causing inconsistent information to be presented to the system error handler.

Programming Note

An example of how failure to satisfy the requirements given above can cause inconsistent information to be presented to the system error handler is as follows. If the value \( X_0 \) (an illegal instruction) is executed, causing the system illegal instruction handler to be invoked, and before the error handler can load \( X_0 \) into a register, \( X_0 \) is replaced with \( X_1 \), an Add Immediate instruction, it will appear that a legal instruction caused an illegal instruction exception.
It is possible to apply a patch or to instrument a given program without the need to suspend or halt the program. This can be accomplished by modifying the example shown in the Programming Note at the end of Section 1.8 where one program is creating instructions to be executed by one or more other programs.

In place of the Store to a flag to indicate to the other programs that the code is ready to be executed, the program that is applying the patch would replace an instruction in the original program with an I-form Branch instruction that would cause any program executing the Branch to branch to the newly created code. The first instruction in the newly created code must be an isync, which will cause any prefetched instructions to be discarded, ensuring that the execution is consistent with the newly created code. The instruction storage location containing the isync instruction in the patch area must be consistent with data storage with respect to the processor that will execute the patched code before the Store which stores the new Branch instruction is performed.

The ability to modify instructions that may be being executed provides opportunities for significant performance benefits in interpretive environments (e.g., Java), and can be used to instrument code dynamically to isolate critical hardware and software bugs.
In this section, "Load instruction" includes the Cache Management and other instructions that are stated in the instruction descriptions to be “treated as a Load”, and similarly for "Store instruction".

The following instructions are never restarted after having accessed any portion of the storage operand (unless the instruction causes a “Data Address Watchpoint match”, for which the corresponding rules are given in Book III).

1. A Store instruction that causes an atomic access
2. A Load instruction that causes an atomic access to storage that is both Caching Inhibited and Guarded

Any other Load or Store instruction may be partially executed and then aborted after having accessed a portion of the storage operand, and then re-executed (i.e., restarted, by the processor or the operating system). If an instruction is partially executed, the contents of registers are preserved to the extent that the correct result will be produced when the instruction is re-executed. Additional restrictions on the partial execution of instructions are described in Section 7.6 of Book III.

--- Programming Note ---

There are many events that might cause a Load or Store instruction to be restarted. For example, a hardware error may cause execution of the instruction to be aborted after part of the access has been performed, and the recovery operation could then cause the aborted instruction to be re-executed.

When an instruction is aborted after being partially executed, the contents of the instruction pointer indicate that the instruction has not been executed, however, the contents of some registers may have been altered and some bytes within the storage operand may have been accessed. The following are examples of an instruction being partially executed and altering the program state even though it appears that the instruction has not been executed.

1. Load Multiple, Load String: Some registers in the range of registers to be loaded may have been altered.
2. Any Store instruction, dcbz: Some bytes of the storage operand may have been altered.

--- Programming Note ---

In order to ensure that the contents of registers are preserved to the extent that a partially executed instruction can be re-executed correctly, the registers that are preserved must satisfy the following conditions. For any given instruction, zero or more of the conditions applies.

- For a fixed-point Load instruction that is not a multiple or string form, if RT=RA or RT=RB then the contents of register RT are not altered.
- For an update form Load or Store instruction, the contents of register RA are not altered.
Chapter 3. Management of Shared Resources

The facilities described in this section provide the means to control the use of resources that are shared with other processors.

3.1 Program Priority Registers

The Program Priority Register (PPR) is a 64-bit register that controls the program’s priority. The PPR provides access to the full 64-bit PPR, and the Program Priority Register 32-bit (PPR32) provides access to the upper 32 bits of the PPR. The layouts of the PPR and PPR32 are shown in Figure 1.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11:13</td>
<td>Program Priority (PRI)</td>
</tr>
<tr>
<td>(PPR3243:45)</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>very low</td>
</tr>
<tr>
<td>010</td>
<td>low</td>
</tr>
<tr>
<td>011</td>
<td>medium low</td>
</tr>
<tr>
<td>100</td>
<td>medium</td>
</tr>
<tr>
<td>101</td>
<td>medium high</td>
</tr>
</tbody>
</table>

Programs can always set the PRI field to very low, low, medium low, and medium priorities; programs may be allowed to set the PRI field to medium high priority during certain time intervals. (See Section 5.3.6.) If the program priority is medium high when the time interval expires or if an attempt is made to set the priority to medium high when it is not allowed, the PRI field is set to medium.

If other values are written to this field, the PRI field is not changed. (See Section 5.3.5 of Book III for additional information.)

All other fields are reserved.

Programming Note

The ability to access the low-order half of the PPR (and thus the use of mfppr and mtppr) might be phased out in a future version of the architecture.

Programming Note

By setting the PRI field, a programmer may be able to improve system throughput by causing system resources to be used more efficiently.

E.g., if a program is waiting on a lock (see Section B.2), it could set low priority, with the result that more processor resources would be diverted to the program that holds the lock. This diversion of resources may enable the lock-holding program to complete the operation under the lock more quickly, and then relinquish the lock to the waiting program.

Programming Note

or Rx,Rx,Rx can be used to modify the PRI field; see Section 3.2.

Programming Note

When the system error handler is invoked, the PRI field may be set to an undefined value.

Figure 1. Program Priority Register
3.2 “or” Instruction

Setting the PPR

The or Rx,Rx,Rx (see Book I) instruction can be used to set PPRPRI as shown in Table 1. or Rx,Rx,Rx does not set PPRPRI.

<table>
<thead>
<tr>
<th>Rx</th>
<th>PPRPRI</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>001</td>
<td>very low</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>low</td>
</tr>
<tr>
<td>6</td>
<td>011</td>
<td>medium low</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>medium</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>medium high</td>
</tr>
</tbody>
</table>

Table 1: Priority levels for or Rx,Rx,Rx

Programs can always set the PRI field to very low, low, medium low, and medium priorities; programs may be allowed to set the PRI field to medium high priority during certain time intervals. (See Section 5.3.6 of Book III.) If the program priority is medium high when the time interval expires or if an attempt is made to set the priority to medium high when it is not allowed, the PRI field is set to medium.

Programming Note

Warning: Other forms of or Rx,Rx,Rx that are not described in this section and in Section 4.3.3 may also cause program priority to change. Use of these forms should be avoided except when software explicitly intends to alter program priority. If a no-op is needed, the preferred no-op (ori 0,0,0) should be used.
Chapter 4.  Storage Control Instructions

4.1 Parameters Useful to Application Programs

It is suggested that the operating system provide a service that allows an application program to obtain the following information.

1. The virtual page sizes
2. Coherence block size
3. Reservation granule size
4. An indication of the cache model implemented (e.g., Harvard-style cache, combined cache)
5. Instruction cache size
6. Data cache size
7. Instruction cache block size
8. Data cache block size
9. Instruction cache associativity
10. Data cache associativity
11. Number of stream IDs supported for the stream variant of dcbt
12. Factors for converting the Time Base to seconds

If the caches are combined, the same value should be given for an instruction cache attribute and the corresponding data cache attribute.

4.2 Data Stream Control Register (DSCR)

The layout of the Data Stream Control Register (DSCR) is shown in Figure 2 below.

Figure 2.  Data Stream Control Register

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>Software Transient Enable (SWTE)</td>
</tr>
<tr>
<td></td>
<td>0  SWTE is disabled.</td>
</tr>
<tr>
<td></td>
<td>1  Applies the transient attribute to software-defined streams.</td>
</tr>
<tr>
<td>40</td>
<td>Hardware Transient Enable (HWTE)</td>
</tr>
<tr>
<td></td>
<td>0  HWTE is disabled.</td>
</tr>
<tr>
<td></td>
<td>1  Applies the transient attribute to hardware-detected streams.</td>
</tr>
<tr>
<td>41</td>
<td>Store Transient Enable (STE)</td>
</tr>
<tr>
<td></td>
<td>0  STE is disabled.</td>
</tr>
<tr>
<td></td>
<td>1  Applies the transient attribute to store streams.</td>
</tr>
<tr>
<td>42</td>
<td>Load Transient Enable (LTE)</td>
</tr>
<tr>
<td></td>
<td>0  LTE is disabled.</td>
</tr>
<tr>
<td></td>
<td>1  Applies the transient attribute to load streams.</td>
</tr>
<tr>
<td>43</td>
<td>Software Unit count Enable (SWUE)</td>
</tr>
<tr>
<td></td>
<td>0  SWUE is disabled.</td>
</tr>
<tr>
<td></td>
<td>1  Applies the unit count to software-defined streams.</td>
</tr>
<tr>
<td>44</td>
<td>Hardware Unit count Enable (HWUE)</td>
</tr>
<tr>
<td></td>
<td>0  HWUE is disabled.</td>
</tr>
<tr>
<td></td>
<td>1  Applies the unit count to hardware-detected streams.</td>
</tr>
<tr>
<td>45:54</td>
<td>Unit Count (UNITCNT)</td>
</tr>
<tr>
<td></td>
<td>Number of units in data stream.</td>
</tr>
<tr>
<td>55:57</td>
<td>Depth Attainment Urgency (URG)</td>
</tr>
<tr>
<td></td>
<td>This field indicates how quickly the prefetch depth should be reached for hardware-detected streams. Values and their meanings are as follows.</td>
</tr>
<tr>
<td></td>
<td>0  default</td>
</tr>
<tr>
<td></td>
<td>1  not urgent</td>
</tr>
<tr>
<td></td>
<td>2  least urgent</td>
</tr>
<tr>
<td></td>
<td>3  less urgent</td>
</tr>
<tr>
<td></td>
<td>4  medium</td>
</tr>
<tr>
<td></td>
<td>5  urgent</td>
</tr>
<tr>
<td></td>
<td>6  more urgent</td>
</tr>
<tr>
<td></td>
<td>7  most urgent</td>
</tr>
<tr>
<td>58</td>
<td>Load Stream Disable (LSD)</td>
</tr>
<tr>
<td></td>
<td>0  No effect.</td>
</tr>
<tr>
<td></td>
<td>1  Disables hardware detection and initiation of load streams.</td>
</tr>
</tbody>
</table>
59  **Stride-N Stream Enable** (SNSE)
0  No effect.
1  Enables the hardware detection and initiation of load and store streams that have a stride greater than a single cache block. Such load streams are detected only when LSD is also zero. Such store streams are detected only when SSE is also one.

60  **Store Stream Enable** (SSE)
0  No effect.
1  Enables hardware detection and initiation of store streams.

61:63  **Default Prefetch Depth** (DPFD)
This field supplies a prefetch depth for hardware-detected streams and for software-defined streams for which a depth of zero is specified or for which `dcbt/dcbtst` with TH=1010 is not used in their description. Values and their meanings are as follows.
0  default (LPCR_{DPFD})
1  none
2  shallowest
3  shallow
4  medium
5  deep
6  deeper
7  deepest

The contents of the DSCR affect how a processor handles hardware-detected and software-defined data streams. The DSCR provides the only means by which software can control or supply information for hardware-detected data streams. The DPFD, UNITCNT, and transient fields may also be used instead of the TH=01010 variant of `dcbt` for software-defined data streams, especially when multiple streams have these attributes in common. See Section 4.3.2, “Data Cache Instructions” on page 1053, for information on streams and how software may specify them.

--- Programming Note ---

The purpose of Depth Attainment Urgency is to regulate the rate of prefetch generation from the cycle at which the hardware first detects an incipient stream until the cycle when the prefetch Depth is reached. A more urgent setting will benefit applications that are dominated by short to medium length streams, because otherwise prefetching does not occur rapidly enough to benefit them. In contrast, applications that frequently cause unproductive prefetches due to stream mispredictions will benefit from a less urgent setting.

Unlike the Depth, the Depth Attainment Urgency applies only to hardware-detected streams. Furthermore, the DSCR provides the only point of control for this parameter. Software-defined streams are assumed not to have the correctness risk associated with hardware streams, and therefore are set to reach their depth relatively quickly.

--- Programming Note ---

In versions of the architecture that precede Version 2.07, `mtspr` specifying the DSCR caused all active and nascent data streams to cease to exist. In those versions of the architecture, the DSCR was used as an overall control mechanism to specify a single global profile for all streams. Beginning with Version 2.07, the DSCR is intended to control and accelerate the creation of new streams without disturbing existing streams.

--- Programming Note ---

The URG, LSD, SNSE and SSE fields do not affect the initiation of streams specified using the `dcbt` and `dcbtst` instructions.

Note that even when SNSE is not set, hardware may detect Stride-N streams in intervals when they access elements that map to sequential cache blocks.

--- Programming Note ---

In order for the DSCR to apply the transient attribute to streams, at least two of the four enable bits must be set: one to choose a type of access (load or store), and one to choose a kind of prefetching (software-defined or hardware-detected).
4.3 Cache Management Instructions

The Cache Management instructions obey the sequential execution model except as described in Section 4.3.1.

In the instruction descriptions the statements “this instruction is treated as a Load” and “this instruction is treated as a Store” mean that the instruction is treated as a Load (Store) from (to) the addressed byte with respect to address translation, the definition of program order on page 1029, storage protection, reference and change recording, the storage access ordering described in Section 1.7.1, and Performance Monitor events (see Section 10.4.5 of Book III).

Some Cache Management instructions contain a CT field that is used to specify a cache level within a cache hierarchy or a portion of a cache structure to which the instruction is to be applied. The correspondence between the CT value specified and the cache level is shown below.

<table>
<thead>
<tr>
<th>CT Field Value</th>
<th>Cache Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Primary Cache</td>
</tr>
<tr>
<td>2</td>
<td>Secondary Cache</td>
</tr>
</tbody>
</table>

CT values not shown above may be used to specify implementation-dependent cache levels or implementation-dependent portions of a cache structure.
### 4.3.1 Instruction Cache Instructions

**Instruction Cache Block Invalidate X-form**

\[ \text{icbi RA, RB} \]

Let the effective address (EA) be the sum \((RA|0)+(RB)\).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of any processors, the block is invalidated in those instruction caches.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the instruction cache of this processor, the block is invalidated in that instruction cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a Load (see Section 4.3), except that reference and change recording need not be done.

**Special Registers Altered:**

None

---

**Programming Note**

Because the instruction is treated as a Load, the effective address is translated using translation resources that are used for data accesses, even though the block being invalidated was copied into the instruction cache based on translation resources used for instruction fetches (see Book III).

---

**Programming Note**

The invalidation of the specified block need not have been performed with respect to the processor executing the icbi instruction until a subsequent isync instruction has been executed by that processor. No other instruction or event has the corresponding effect.

---

**Instruction Cache Block Touch X-form**

\[ \text{icbt CT, RA, RB} \]

Let the effective address (EA) be the sum \((RA|0)+(RB)\).

The icbt instruction provides a hint that the program will probably soon execute code from the block containing the byte addressed by EA, and that the block containing the byte addressed by EA is to be loaded into the cache specified by the CT field. (See Section 4.3 of Book II.) If the CT field is set to a value not supported by the implementation, no operation is performed.

The hint is ignored if the block is Caching Inhibited.

This instruction treated as a Load (see Section 4.3), except that the system data storage error handler is not invoked, and reference and change recording need not be done.

**Special Registers Altered:**

None
4.3.2 Data Cache Instructions

The Data Cache instructions control various aspects of the data cache.

**TH field in the dcbt and dcbtst instructions**

Described below are the TH field values for the dcbt and dcbtst instructions. For all TH field values which are not listed, the hint provided by the instruction is undefined.

**TH=0b00000**

If TH=0b00000, the dcbt/dcbtst instruction provides a hint that the program will probably soon access the block containing the byte addressed by EA.

**TH=0b01000 - 0b01111**

The dcbt/dcbtst instructions provide hints regarding a sequence of accesses to data elements, or indicate the expected use thereof. Such a sequence is called a “data stream”, and a dcbt/dcbtst instruction in which TH is set to one of these values is said to be a “data stream variant” of dcbt/dcbtst. In the remainder of this section, “data stream” may be abbreviated to “stream”.

A data stream to which a program may perform Load accesses is said to be a “load data stream”, and is described using the data stream variants of the dcbt instruction. A data stream to which a program may perform Store accesses is said to be a “store data stream”, and is described using the data stream variants of the dcbtst instruction.

When, and how often, effective addresses for a data stream are translated is implementation-dependent.

Each data element is associated with a unit of storage, which is the aligned 128-byte location in storage that contains the first byte of the element. The data stream variants may be used to specify the address of the beginning of the data stream, the displacement (stride) between the first byte of successive elements, and the number of unique units of storage that are associated with all of the data elements. If the stride is specified, both the stride and the address of the first element are specified at 4 byte granularity. If the stride is not specified, the address of the first element is the address of the first unit.

**Programming Note**

The architecture does not provide a way to specify the size of the data elements that compose a stream. An implementation may assume some fixed size for all data elements. As a result, depending on the offset, stride, and size (and in particular whether the elements are aligned), the implementation may reduce the latency for accessing only a portion of some of the elements. A future version of the architecture may enable the specification of element size to avoid this limitation.

Each such data stream is associated, by software, with a stream ID, which is a resource that the processor uses to distinguish the data stream from other such data streams. The number of stream IDs is an implementation-dependent value in the range 1:16. Stream IDs are numbered sequentially starting from 0.

**Programming Note**

The number of stream IDs available for a program to use may be dependent on the current degree of multithreading of a processor in the system.

The encodings of the TH field and of the corresponding EA values are as follows. In the EA layout diagrams, fields shown as “/”s are reserved. These reserved fields are treated in the same manner as the corresponding case for instruction fields (see Section 1.3.3 of Book I). If a reserved value is specified for a defined EA field, or if a TH value is specified that is not explicitly defined below, the hint provided by the instruction is undefined.

<table>
<thead>
<tr>
<th>TH</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>The dcbt/dcbtst instruction provides a hint that describes certain attributes of a data stream, and may indicate that the program will probably soon access the stream. The EA is interpreted as follows.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:56</td>
<td>EATRUNC</td>
</tr>
<tr>
<td>57</td>
<td>Direction (D)</td>
</tr>
</tbody>
</table>

High-order 57 bits of the effective address of the first element of the data stream. (i.e., the effective address of the first unit of the stream is EATRUNC || 0)
58 Unlimited/GO (UG)
   0 No information is provided by the UG field.
   1 The number of elements in the data stream is unlimited, the elements are adjacent to each other, the program’s need for each element of the stream is not likely to be transient, and the program will probably soon access the stream.

59 Reserved

60:63 Stream ID (ID)
Stream ID to use for this data stream.

01010 The dcbt/dcbtst instruction provides a hint that describes certain attributes of a data stream, or indicates that the program will probably soon access data streams that have been described using data stream variants of the dcbt/dcbtst instruction, or will probably no longer access such data streams.

The EA is interpreted as follows. If GO=1 and $S \neq 0b00$ the hint provided by the instruction is undefined; the remainder of this instruction description assumes that this combination is not used.

<table>
<thead>
<tr>
<th>Bit(s) Description</th>
<th>0:31</th>
<th>32</th>
<th>35:38</th>
<th>39:46</th>
<th>47:56</th>
<th>57</th>
<th>59:63</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO</td>
<td>0 No information is provided by the GO field.</td>
<td>1 For dcbt, the program will probably soon access all nascent load and store data streams that have been completely described, and will probably no longer access all other nascent load and store data streams. All other fields of the EA are ignored. (&quot;Nascent&quot; and &quot;completely described&quot; are defined below.) For dcbt, this field value holds no meaning and is treated as though it were zero.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32:34 Stop (S)</td>
<td>00 No information is provided by the S field.</td>
<td>01 Reserved</td>
<td>10 The program will probably no longer access the data stream (if any) associated with the specified stream ID. (All other fields of the EA except the ID field are ignored.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

35 Reserved

36:38 Depth (DEP)
The DEP field provides a relative estimate of how many elements ahead of the point of stream use the latency-reducing actions should go. This value reflects a comparison of the rate of consumption of the elements of the data stream and the latency to bring an arbitrary element of the stream into cache. The values are as follows.

| Depth | 0 default = DSCR$_{DPFD}$ | 1 none | 2 shallowest | 3 shallow | 4 medium | 5 deep | 6 deeper | 7 deepest |

39:46 Reserved

47:56 UNITCNT
Number of units in data stream.

57 Transient (T)
If T=1, the program’s need for each element of the data stream is likely to be transient (i.e., the time interval during which the program accesses the element is likely to be short).

58 Unlimited (U)
If U=1, the number of units in the data stream is unlimited (and the UNITCNT field is ignored).

59 Reserved

60:63 Stream ID (ID)
Stream ID to use for this data stream (GO=0 and S=0b00), or stream ID associated with the data stream which the program will probably no longer access (S=0b10).
### Programming Note

To maximize the utility of the Depth control mechanism, the architecture provides a hierarchy of three ways to program it. The DPFD field in the LPCR is used by the provisory/firmware to set a safe or appropriate default depth for unaware operating systems and applications. The DPFD field in the DSCR may be initialized by the aware OS and overwritten by an application via the OS-provided service when per stream control is unnecessary or unaffordable. The DEP field in the EA specification when TH=0b01010 may be used by the application to specify the depth on a per-stream basis.

The number of elements ahead of the point of stream use indicated by a given depth value may differ across implementations, as may the latency to bring a given element into the cache. To achieve optimum performance, some experimentation with different depth values may be necessary.

### Table: Bit(s) Description

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Reserved</td>
</tr>
<tr>
<td>32:49</td>
<td>Stride</td>
</tr>
<tr>
<td>50</td>
<td>Reserved</td>
</tr>
<tr>
<td>51:55</td>
<td>Offset</td>
</tr>
<tr>
<td>60:63</td>
<td>Stream ID (ID)</td>
</tr>
</tbody>
</table>

The displacement, in words, between the first byte of successive elements in the stream. The effective address of the Nth element in the stream is

\[(N-1) \times \text{STRIDE} \times 4\]

greater than or less than the effective address of the first element of the stream, depending on the direction specified for the stream.

If the specified stream ID value is greater than m-1, where m is the number of stream IDs provided by the implementation, and either (a) TH=0b01000 or TH=0b01011, or (b) TH=0b01010 with GO=0 and S\(\neq 0b11\), no hint is provided by the instruction.

The following terminology is used to describe the state of a data stream. Except as described in the paragraph after the next paragraph, the state of a data stream at a given time is determined by the most recently provided hint(s) for the stream.

- A data stream for which only descriptive hints have been provided (by `dcbt/dcbtst` instructions with TH=0b01000 and UG=0, TH=0b01010 and GO=0 and S=0b00, and/or with TH=0b01011) is said to be "nascent". A nascent data stream for which all relevant descriptive hints have been provided (by the `dcbt/dcbtst` usages listed in the preceding sentence) is considered to be "completely described". The order of descriptive hints with respect to one another is unimportant.

- A data stream for which a hint has been provided (by a `dcbt/dcbtst` instruction with TH=0b01000 and UG=1 or `dcbt` with TH=0b01010 and GO=1) that the program will probably soon access it is said to be "active".

- A data stream that is either nascent or active is considered to "exist".

- A data stream for which a hint has been provided (e.g., by a `dcbt` instruction with TH=0b01010 and S\(\neq 0b00\)) that the program will probably no longer access it is considered no longer to exist.

The hint provided by a `dcbt/dcbtst` instruction with TH=0b01000 and UG=1 implicitly includes a hint that the program will probably no longer access the data stream (if any) previously associated with the specified stream ID. The hint provided by a `dcbt/dcbtst` instruction with TH=0b01000 and UG=0, or with TH=0b01010 and GO=0 and S=0b00, or with TH=0b01011 implicitly includes a hint that the program will probably no longer access the active data stream (if any) previously associated with the specified stream ID.

If a data stream is specified without using a `dcbt/dcbtst` instruction with TH=0b01010 and GO=0 and S=0b00, then the number of elements in the stream is unlimited, and the program's need for each element of the stream is not likely to be transient. If a data stream is specified without using a `dcbt/dcbtst` instruction with
TH=0b01011, then the stream will access consecutive units of storage.

A context switch on a particular thread causes all existing data streams for that thread to cease to exist. The mechanism of detection of a context switch is implementation-dependent. In addition, depending on the implementation, certain conditions and events may cause an existing data stream to cease to exist; for example, in some implementations an existing data stream ceases to exist when it comes to the end of a page.
Programming Note

To obtain the best performance across the widest range of implementations that support the data stream variants of `dcbt/dcbtst`, the programmer should assume the following model when using those variants.

- The processor's response to a hint that the program will probably soon access a given data stream is to take actions that reduce the latency of accesses to the first few elements of the stream. (Such actions may include prefetching cache blocks into levels of the storage hierarchy that are "near" the processor.) Thereafter, as the program accesses each successive element of the stream, the processor takes latency-reducing actions for additional elements of the stream, pacing these actions with the program's accesses (i.e., taking the actions for only a limited number of elements ahead of the element that the program is currently accessing).

- The processor's response to a hint that the program will probably no longer access a given data stream, or to the cessation of existence of a data stream, is to stop taking latency-reducing actions for the stream.

- A data stream having finite length ceases to exist when the latency-reducing actions have been taken for all elements of the stream.

- If the program ceases to need a given data stream before having accessed all elements of the stream (always the case for streams having unlimited length), performance may be improved if the program then provides a hint that it will no longer access the stream (e.g., by executing the appropriate `dcbt` instruction with TH=0b01010 and S≠0b00).

- At each level of the storage hierarchy that is "near" the processor, elements of a data stream that is specified as transient are most likely to be replaced. As a result, it may be desirable to stagger addresses of streams (choose addresses that map to different cache congruence classes) to reduce the likelihood that an element of a transient stream will be replaced prior to being accessed by the program.

- Processors that comply with versions of the architecture that do not support the TH field at all treat TH = 0b01000, 0b01010, and 0b01011 as if TH = 0b00000.

- A single set of stream IDs is shared between the `dcbt` and `dcbtst` instructions.

- On some implementations, data streams that are not specified by software may be detected by the processor. Such data streams are called "hardware-detected data streams". On some such implementations, data stream resources (resources that are used primarily to support data streams) are shared between software-specified data streams and hardware-detected data streams. On these latter implementations, the programming model includes the following.
  - Software-specified data streams take precedence over hardware-detected data streams in use of data stream resources.
  - The processor's response to a hint that the program will probably no longer access a given data stream, or to the cessation of existence of a data stream, includes releasing the associated data stream resources, so that they can be used by hardware-detected data streams.
The latency-reducing actions taken in response to a program’s hints about access to a data stream, including the depth and urgency parameters, may vary based on its behavior and on the behavior of other programs sharing platform resources, as well as on the design of the platform resources they use. Without actually changing the stream specification or DSCR parameters, the processor may adjust its actions (e.g. slow down prefetches or be more selective choosing them) based on their effectiveness and on the availability of storage bandwidth. In general, the goal of this variation is to improve overall system performance and fairness across the set of programs that share resources. There often will be a performance benefit, however, from adjusting stream specifications to the platform and co-resident programs to adjust for these actions by the processor.
This Programming Note describes several aspects of using the data stream variants of the `dcbt` and `dcbtst` instructions.

- A non-transient data stream having unlimited length and which will access consecutive units in storage can be completely specified, including providing the hint that the program will probably soon access it, using one `dcbt` instruction. The corresponding specification for a data stream having other attributes requires two or three `dcbt/dcbtst` instructions to describe the stream and one additional `dcbt` instruction to start the stream. However, one `dcbt` instruction with TH=0b01010 and GO=1 can apply to a set of the data streams described in the preceding sentence, so the corresponding specification for n such data streams requires 2×n to 3×n `dcbt/dcbtst` instructions plus one `dcbt` instruction. (There is no need to execute a `dcbt/dcbtst` instruction with TH=0b01010 and S=0b10 for a given stream ID before using the stream ID for a new data stream; the implicit portion of the hint provided by `dcbt/dcbtst` instructions that describe data streams suffices.)

- If it is desired that the hint provided by a given `dcbt/dcbtst` instruction be provided in program order with respect to the hint provided by another `dcbt/dcbtst` instruction, the two instructions must be separated by an `eieio` instruction. For example, if a `dcbt` instruction with TH=0b01010 and GO=1 is intended to indicate that the program will probably soon access nascent data streams described (completely) by preceding `dcbt/dcbtst` instructions, and is intended not to indicate that the program will probably soon access nascent data streams described (completely) by following `dcbt/dcbtst` instructions, an `eieio` instruction must separate the `dcbt` instruction with GO=1 from the preceding `dcbt/dcbtst` instructions, and another `eieio` instruction must separate that `dcbt` instruction from the following `dcbt/dcbtst` instructions.

- In practice, the second `eieio` described above can sometimes be omitted. For example, if the program consists of an outer loop that contains the `dcbt/dcbtst` instructions and an inner loop that contains the `Load` or `Store` instructions that access the data streams, the characteristics of the inner loop and of the implementation's branch prediction mechanisms may make it highly unlikely that hints corresponding to a given iteration of the outer loop will be provided out of program order with respect to hints corresponding to the previous iteration of the outer loop. (Also, any providing of hints out of program order affects only performance, not program correctness.)

- To mitigate the effects of context switches on data streams, it may be desirable to specify a given "logical" data stream as a sequence of shorter, component data streams. Similar considerations apply to conditions and events that, depending on the implementation, may cause an existing data stream to cease to exist; for example, in some implementations an existing data stream ceases to exist when it comes to the end of a virtual page.

- If it is desired to specify data streams without regard to the number of stream IDs provided by the implementation, stream IDs should be assigned to data streams in order of decreasing stream importance (stream ID 0 to the most important stream, stream ID 1 to the next most important stream, etc.). This order ensures that the hints for the most important data streams will be provided.

- In practice, it is usually the case that it is preferable for data streams of the main or parent application running on a particular thread to be prioritized over data streams of a leaf or child application or function call. As an example, a library function call would usually fall under the category of a leaf function whose data streams should be treated with lower priority than those of the parent. As such, it is advised that programmers writing library functions or other functions which are expected to have leaf routing use the lowest stream priority possible. For example, if a leaf-type function instantiates two data streams, best practice is to use stream ID 14 for its most important data stream, and stream ID 15 for its second most important stream.

**Version 3.1**

**Chapter 4. Storage Control Instructions**

---

**TH=0b10000**

If TH=0b10000, the `dcbt` instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA, and that the program’s need for the block will be transient (i.e., the time interval during which the program accesses the block is likely to be short).
TH=0b10001

If TH=0b10001, the $dcbt$ instruction provides a hint that the program will probably not access the block containing the byte addressed by EA for a relatively long period of time.
**Data Cache Block Touch X-form**

\[ \text{dcbt RA,RB,TH} \]

Let the effective address (EA) be the sum \((RA|0)+(RB)\).

The `dcbt` instruction provides a hint that describes a block or data stream to which the program may perform a Load access. The instruction is also used to indicate imminent access or end of access to described load and store data streams. A hint that the program will probably soon load from a given storage location is ignored if the location is Caching Inhibited or Guarded.

The only operation that is “caused” by the `dcbt` instruction is the providing of the hint. The actions (if any) taken by the processor in response to the hint are not considered to be “caused by” or “associated with” the `dcbt` instruction (e.g., `dcbt` is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by memory barriers.

The `dcbt` instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified at the beginning of this section. If \(TH\neq 0b01010\) and \(TH\neq 0b01011\), this instruction is treated as a Load (see Section 4.3), except that the system data storage error handler is not invoked, and reference and change recording need not be done.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Extended mnemonics are provided for the `Data Cache Block Touch` instruction so that it can be coded with the TH value as the last operand for all categories, and so that the transient hint can be specified without coding the TH field explicitly.

**Extended:**

- `dcbct RA,RB,TH`
- `dcbtds RA,RB,TH`
- `dcbtt RA,RB`
- `dcbna RA,RB`

**Equivalent to:**

- `dcbt for TH values of 0b00000 - 0b00111`
- `dcbt for TH values of 0b00000 or 0b01000 - 0b01111`
- `dcbt for TH value of 0b10000`
- `dcbt for TH value of 0b10000`

New programs should avoid using the `dcbt` and `dcbtst` mnemonics; one of the extended mnemonics should be used exclusively.

If the `dcbt` mnemonic is used with only two operands, the TH operand is assumed to be 0b00000.

Processors that comply with versions of the architecture that precede Version 2.01 do not necessarily ignore the hint provided by `dcbt` and `dcbtst` if the specified block is in storage that is Guarded and not Caching Inhibited.

See the Programming Notes at the beginning of this section.
**Data Cache Block Touch for Store X-form**

```
dcbtst RA,RB,TH
```

Let the effective address (EA) be the sum (RA|0)+(RB).

The `dcbtst` instruction provides a hint that describes a block or data stream to which the program may perform a Store access, or indicates the expected use thereof. A hint that the program will soon store to a given storage location is ignored if the location is Caching Inhibited or Guarded.

The only operation that is “caused by” the `dcbtst` instruction is the providing of the hint. The actions (if any) taken by the processor in response to the hint are not considered to be “caused by” or “associated with” the `dcbtst` instruction (e.g., `dcbtst` is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by memory barriers.

The `dcbtst` instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified at the beginning of this section. If TH=0b00000 or TH=0b01000, this instruction is treated as a `Store` (see Section 4.3), except that the system data storage error handler is not invoked, reference recording need not be done, and change recording is not done.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Extended mnemonics are provided for the `Data Cache Block Touch for Store` instruction so that it can be coded with the TH value as the last operand for all categories, and so that the transient hint can be specified without coding the TH field explicitly.

**Extended:**

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcbtstct RA,RB,TH</td>
<td>dcbtst for TH values of 0b00000 or 0b00000 - 0b00111; other TH values are invalid.</td>
</tr>
<tr>
<td>dcbtstds RA,RB,TH</td>
<td>dcbtst for TH values of 0b00000 or 0b01000 - 0b01111; other TH values are invalid.</td>
</tr>
<tr>
<td>dcbtstt RA,RB</td>
<td>dcbtst for TH value of 0b10000.</td>
</tr>
</tbody>
</table>

**Programming Note**

See the Programming Notes at the beginning of this section.
**Data Cache Block set to Zero X-form**

\[ \text{dcbz \ RA, RB} \]

\[
\begin{array}{cccccc}
0 & 31 & // & 8 & 11 & 16 & 21 & 1014 & / 31
\end{array}
\]

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)
\( EA \leftarrow b + (RB) \)
\( n \leftarrow \text{block size (bytes)} \)
\( m \leftarrow \log_2(n) \)
\( ea \leftarrow E_{0:63-m} \| n_0 \)
\( \text{MEM}(ea, n) \leftarrow 0x00 \)

Let the effective address (EA) be the sum (RA\(0\)+RB).

All bytes in the block containing the byte addressed by EA are set to zero.

This instruction is treated as a Store (see Section 4.3).

**Special Registers Altered:**
- None

**Programming Note**

\[ \text{dcbz} \] does not cause the block to exist in the data cache if the block is in storage that is Caching Inhibited.

For storage that is neither Write Through Required nor Caching Inhibited, \text{dcbz} provides an efficient means of setting blocks of storage to zero. It can be used to initialize large areas of such storage, in a manner that is likely to consume less memory bandwidth than an equivalent sequence of Store instructions.

For storage that is either Write Through Required or Caching Inhibited, \text{dcbz} is likely to take significantly longer to execute than an equivalent sequence of Store instructions. For example, on some implementations dcbz for such storage may cause the system alignment error handler to be invoked; on such implementations the system alignment error handler sets the specified block to zero using Store instructions.

See Section 6.9.1 of Book III for additional information about \text{dcbz}.

---

**Data Cache Block Store X-form**

\[ \text{dcbst \ RA, RB} \]

\[
\begin{array}{cccccc}
0 & 31 & // & 8 & 11 & 16 & 21 & 54 & / 31
\end{array}
\]

Let the effective address (EA) be the sum (RA\(0\)+RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a Load (see Section 4.3), except that reference and change recording need not be done.

**Special Registers Altered:**
- None

**Programming Note**

\[ \text{Data Cache Block Store to Persistent Storage} \] is encoded as a variant of \text{Data Cache Block Flush}. This was necessary so that the instruction has the correct behavior on processors that implement Version 3.0C. The extended mnemonic (\text{dcbstps}) indicates the intended function. (There is expected to be no interest in attaching persistent storage to processors that comply with versions of the architecture that precede Version 3.0C.)
Data Cache Block Flush X-form

dcbf RA, RB, L

Let the effective address (EA) be the sum (RA|0)+(RB).

L=0

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data cache of this processor.

L=1 (“dcbf local”)

The L=1 form of the dcbf instruction permits a program to limit the scope of the “flush” operation to the data cache of this processor. If the block containing the byte addressed by EA is in the data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

L = 3 (“dcbf local primary”)

The L=3 form of the dcbf instruction permits a program to limit the scope of the “flush” operation to the primary data cache of this processor. If the block containing the byte addressed by EA is in the primary data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

L = 4 (“data cache block flush to persistent storage”)

The L=4 form of the dcbf instruction performs all of the functions of dcbf with L=0. After all writes to main storage, caused by these functions, have updated main storage, if the block maps to main storage that is backed by persistent storage then all locations in the block in main storage that are considered to be modified relative to persistent storage are written to persistent storage and additional locations in the block in main storage may be written to persistent storage.

L = 6 (“data cache block store to persistent storage”)

The L=6 form of the dcbf instruction performs all of the functions of dcbst. After all writes to main storage, caused by these functions, have updated main storage, if the block maps to main storage that is backed by persistent storage then all locations in the block in main storage that are considered to be modified relative to persistent storage are written to persistent storage and additional locations in the block in main storage may be written to persistent storage.

Programming Note

This form of the dcbf instruction is considered to be a functional extension of dcbst, and its extended mnemonic reflects that association.

For the L operand, the values 2, 5, and 7 are reserved. The results of executing a dcbf instruction with L=2, 5, or 7 are boundedly undefined.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a Load (see Section 4.3), except that reference and change recording need not be done.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the Data Cache Block Flush instruction so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand. These are shown as examples with the instruction. See Appendix A. “Assembler Extended Mnemonics” on page 1105. The extended mnemonics are shown below.

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcbf RA, RB</td>
<td>dcbf RA, RB, 0</td>
</tr>
<tr>
<td>dcbfl RA, RB</td>
<td>dcbf RA, RB, 1</td>
</tr>
<tr>
<td>dcbflp RA, RB</td>
<td>dcbf RA, RB, 3</td>
</tr>
<tr>
<td>dcbfps RA, RB</td>
<td>dcbf RA, RB, 4</td>
</tr>
<tr>
<td>dcbstps RA, RB</td>
<td>dcbf RA, RB, 6</td>
</tr>
</tbody>
</table>

Except in the dcbf instruction description in this section, references to “dcbf” in Books I-III imply L=0 unless otherwise stated or obvious from context; “dcbfl” is used for L=1, “dcbflp” is used for L=3, “dcbfps” is used for L=4, and “dcbstps” is used for L=6.
4.3.2.1 Obsolete Data Cache Instructions

The Data Stream Touch (dst), Data Stream Touch for Store (dstt), and Data Stream Stop (dss) instructions (primary opcode 31, extended opcodes 342, 374, and 822 respectively), which were proposed for addition to the Power ISA and were implemented by some processors, must be treated as no-ops (rather than as illegal instructions).

The treatment of these instructions is independent of whether other Vector instructions are available (i.e., is independent of the contents of MSRVEC (see Book III).

Programming Note

These instructions merely provided hints, and thus were permitted to be treated as no-ops even on processors that implemented them.

The treatment of these instructions is independent of whether other Vector instructions are available because, on processors that implemented the instructions, the instructions were available even when other Vector instructions were not.

The extended mnemonics for these instructions were dstt, dsttt, and dssall.
4.3.3 “or” Instruction

“or” Cache Control Hint

This form of or provides a hint that stores caused by preceding Store and dcbz instructions should be performed with respect to other processors and mechanisms as soon as is feasible.

Extended Mnemonics:

Additional extended mnemonic for the or hint:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>miso</td>
<td>or 26,26,26</td>
</tr>
</tbody>
</table>

“miso” is short for “make it so.”

Programming Note

Warning: Other forms of or Rx,Rx,Rx that are not described in this section and in Section 3.2 may also cause program priority to change. Use of these forms should be avoided except when software explicitly intends to alter program priority. If a no-op is needed, the preferred no-op (ori 0,0,0) should be used.

Extended: Equivalent to:
miso or 26,26,26

This form of the or instruction can be used to reduce latency in producer-consumer applications by requesting that modified data be made visible to other processors quickly. In this example it is assumed that the base register is GPR3.

Producer:

```
addi r1,r0,0x1234
sth r1,0x1000(r3) # store data value 0x1234
lwsync # order store before flag store
addi r2,r0,0x0001
sth r2,0x1002(r3) # store nonzero flag byte
or r26,r26,r26 # miso
p_loop:
  lbz r2,0x1002(r3) # load flag byte
  andi r2,r2,0x00FF
  bne p_loop # wait for consumer to clear flag
```

Consumer:

```
c_loop:
  lbz r2,0x1002(r3) # load flag byte
  andi r2,r2,0x00FF
  beq c_loop # wait for producer to set flag to nonzero
  lwsync # order load before data load
  lhz r1,0x1000(r3) # load data value
  lwsync # order load before flag store
  addi r2,r0,0x0000
  stb r2,0x1002(r3) # clear flag byte
  or r26,r26,r26 # miso
```
4.4 Copy-Paste Facility

The Copy-Paste Facility provides a means to copy a block of data from one storage location to another. The most straightforward application is memcpy. Depending on the platform configuration, other alternatives such as posting a work element to an accelerator’s queue or copying a block to another system may be supported. The facility uses pairs of instructions, copy followed by paste, to define the data transfers. (See Section 1.7.1.1, “Storage Ordering of Copy/Paste-Initiated Data Transfers” for the memory model characteristics of these data transfers.)

The requirements for use of the platform-specific functionality may vary across implementations. The specific details are beyond the scope of the Architecture. Authority to use an accelerator is generally established through a call to the hypervisor. The format of the work element is accelerator-specific. Mappings for transfers between systems are generally managed by the operating system. Each transfer preserves the order of bytes in storage and is not affected by the endian mode of the processor.

Since the buffer that holds the block until a data transfer is performed is hidden state (cannot be saved and restored) and there is no way to save the state of the copy, any disruption of program execution (e.g. interrupts, event-based branch) has the potential to prevent the data transfer from completing correctly. The software that handles the disruption is responsible for executing cpabort to clear the state associated with an outstanding data transfer if it will use the Copy-Paste Facility itself or transfer control to another program that might use the facility prior to returning control to the original program.

Correct use of the Copy-Paste Facility consists of a series of copy/paste pairs. The two instructions in a pair need not be adjacent in the instruction stream. Two or more copy instructions with no intervening paste produces a “copy-paste sequence error.” Similarly, a bare paste, with no preceding copy, produces a copy-paste sequence error. Copy-paste sequence errors are reported by the paste for the malformed sequence of instructions.

Programming Note

A paste instruction is ordered with respect to its preceding copy by a dependency on the copy buffer. No explicit synchronization or barrier is required.

Programming Note

WARNING: In rare circumstances, paste, may falsely report successful completion when the copy-paste sequence is coded incorrectly. This may occur if the instruction sequence includes a redundant copy and the sequence is interrupted just prior to the redundant copy. Since interrupts should be rare, any sequence that returns a false positive CR0 value should fail for most executions.

Programming Note

It is always best to avoid unnecessary instructions between the copy and the paste.

Successful transfers are indicated when paste returns 0b001x in CR0. Transient errors (a copy-paste sequence error, a memory management state change (tlbi[fl]) during the transfer, or an implementation-specific transient problem) are indicated by a CR0 value of 0b000x, indicating the sequence should be retried. A sequence error is considered transient because it could have been caused by an interruption between the copy and paste.) Fatal errors unique to the Copy-Paste Facility (attempting to copy from an accelerator or attempting to use an accelerator that has not been properly configured) cause the system data storage error handler to be invoked when the (associated) paste instruction is executed. paste instructions that cause or report transient errors, fatal errors unique to the Copy-Paste Facility, or successful transfer completion reset the state of the facility so that a subsequent copy-paste sequence can begin with a clean slate.

Programming Note

For paste, to address space mapped to another system, OS control over that mapping is deemed to be a sufficient check on the configuration of the channel to the other system, so that a unique data storage interrupt type is not required.
The Copy-Paste Facility is the only means to address an accelerator. If any other storage access (implicit or explicit, instruction or data) addresses an accelerator, a Machine Check exception will result. Unlike other Machine Check exceptions, this one will generally be presented with ordering and priority similar to that for a storage protection exception.

### Copy X-form

copy RA,RB

```
0 31 10 16 21 31
31 6 9 10 11 16 21
```

if RA = 0 then b ← 0
else
   b ← (RA)

EA ← b + (RB)

copy_buffer ← memory(EA,128) || MEMmetadata(EA,128)

Let the effective address (EA) be the sum (RA|0)+(RB).

The 128 bytes in storage, and associated metadata, addressed by EA is loaded into the copy buffer.

If the EA is not a multiple of 128, the system alignment error handler is invoked.

If the specified block is in storage that is Caching Inhibited, the system data storage error handler is invoked.

When successful, this instruction is treated as a Load (see Section 4.3, "Cache Management Instructions"), except that the data transfer ordering is described in Section 1.7.1.1, "Storage Ordering of Copy/Paste-Initiated Data Transfers".

### Paste X-form

paste. RA,RB,L

```
0 31 6 9 10 11 16 21 31
31 6 9 10 11 16 21
```

if there was a copy-paste sequence error or a translation conflict
   CR0 ← 0b000 || XER_S0
else
   if RA = 0 then b ← 0
   else
      b ← (RA)
   EA ← b + (RB)
   if L=1 then
      copy_buffer.md ← 0 /* clear metadata in buffer
   wait for completion status
   if there was a data transfer problem
      CR0 ← 0b000 || XER_S0
   else
      CR0 ← 0b001 || XER_S0
   clear the state of the Copy-Paste Facility

If there was a copy-paste sequence error or a translation conflict, set CR0 to indicate failure. Otherwise, continue as follows.

Let the effective address (EA) be the sum (RA|0)+(RB).

If L=1, then set metadata bits in the copy buffer to zero.

The physical target of the operation, and by implication the function to be performed, is determined by the real address that is translated from EA. If the real address is in the platform’s system memory, a simple copy is performed. If the real address has an associated mapping to another system, the copy buffer is transmitted to the other system. If the real address is control memory for an accelerator, the contents of the copy buffer is queued to the accelerator. There is a wait for completion status on the data transfer. CR0 is set as follows based on the completion status.

<table>
<thead>
<tr>
<th>CR0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td></td>
</tr>
<tr>
<td>0b001</td>
<td></td>
</tr>
</tbody>
</table>

The state of the Copy-Paste Facility is cleared.

If MSR_PP=1, paste. with L=0 is an invalid form.

If the EA is not a multiple of 128, the system alignment error handler is invoked.

If the specified block is in storage that is Caching Inhibited, the system data storage error handler is invoked.

### Programming Note

A failure of a data transfer may be the result of a shortage of the resources required to complete the operation. (Such failures should only take place for accelerator invocations,) When the resources are known to be shared by multiple programs, a credit-based system is frequently used to improve quality of service. If such a credit system is in use, or if the resources are not shared, the program should continually repeat the copy/paste, pair until it succeeds. However, if no credit system is in use for shared resources, it may be appropriate to apply some sort of backoff algorithm after having retried the copy/paste, pair a few times.

### Programming Note

Accelerator address space is to be marked No-execute by the hypervisor, so that an instruction fetch will violate storage protection rather than causing a Machine Check.

### Special Registers Altered:

None
If the associated copy specified an accelerator or the paste. specifies an accelerator that was not properly configured, the data storage error handler will be invoked.

When successful, this instruction is treated as a Store (see Section 4.3, “Cache Management Instructions”), except that the data transfer ordering is described in Section 1.7.1.1, “Storage Ordering of Copy/Paste-Initiated Data Transfers”.

Special Registers Altered:

CR0

Extended Mnemonics:

Extended mnemonic for paste.:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>paste. RA, RB</td>
<td>paste. RA, RB, 1</td>
</tr>
</tbody>
</table>

**Copy-Paste Abort X-form**

cpabort

```
0 31 8 11 16 21 838 31
```

clear the state of the Copy-Paste Facility

The *cpabort* instruction causes a data transfer to fail if one is in progress.

Any pending errors in the Copy-Paste Facility are cleared and the state is reset to prepare for a new *copy*.

**Special Registers Altered:**

None
4.5 Atomic Memory Operations

The Atomic Memory Operation (AMO) facility may be used to optimize performance when many software threads are manipulating shared control structures concurrently. In such situations, accessing the shared data frequently involves transferring the data from one processor’s cache to another. The latency of such transfers can become the limiting factor in the performance of some environments. Rather than moving the data to the work, AMOs move the work to the data. The mental model is of an agent consisting of an execution unit and a work queue near memory that receives atomic update requests from all the processors in the system.

Despite that AMOs are performed at memory, their function is only defined for storage that is not Caching Inhibited. This is done so that software can transparently access the same data using normal loads and stores. But furthermore, AMOs generally behave as typical explicit storage accesses performed by the thread, with respect to storage access ordering. The few complications are described below. Since the performance advantage of AMOs derives from avoiding time of flight through cache hierarchies, software should avoid frequent mixing of normal loads and stores to the same storage locations. AMOs are also restricted to storage that is not Guarded and storage that is not Write Through Required to limit implementation complexity.

The facility specifies a set of atomic update operations that a processor may send, accompanied by operands from GPRs, to the memory to be performed. The operations are expressed using the Load Atomic (LAT) and Store Atomic (STAT) instructions. Each of these instructions performs an atomic update operation (load followed by some manipulation and a store) on some location in storage. As a result, these instructions are considered to be both fixed-point Load instructions and fixed-point Store instructions, and any reference elsewhere in the architecture to fixed-point Load or fixed-point Store instructions apply to these instructions as well, unless otherwise stated or obvious from context. For example, in order to perform an AMO, it is necessary to have both read and write access to the storage location. Another example is that the DAWR will detect a match if either Data Read or Data Write is selected. Yet another example is that a Trace interrupt will indicate both a Load and a Store instruction have been executed. Barrier action will be based on whether the barrier would give a load or a store the stronger ordering. The difference between the Load Atomic instructions and the Store Atomic instructions is simply that the Load Atomic instructions return a result to a GPR, while the Store Atomic instructions do not. In the RTL in the following subsections, the “lat” and “stat” functions represent the manipulations performed by the memory agent. The parameters shown are the maximum storage footprint, the maximum list of registers, and the function code that are provided to the agent. If the specified registers wrap (e.g. RT=R31 and RT+1=R0), the wrapping is permitted. Such an instruction is not an invalid form. Destructive encodings are also permitted (i.e. a LAT specified with RT=RA).

Except in this section, references to “atomic update” in Books I-III imply use of the Load And Reserve and Store Conditional instructions unless otherwise stated or obvious from context.

--- Programming Note

The best performance for the Atomic Memory Operations will be realized when the targeted storage locations are accessed only using AMOs. If it is necessary to perform other i=0 loads and stores to those addresses, the result will still be correct, but performance will suffer. In such circumstances, it is not helpful to performance to flush the data to memory using dcbf.

--- Programming Note

Note that the descriptions of AMO operations are Endian independent. The only effect of Endian on these operations is the obvious one that byte significance within an individual datum reflects the Endian mode.

4.5.1 Load Atomic

The Atomic Loads perform an atomic update to an aligned memory location and return a value to a GPR. The manipulation performed on the memory value and the value that is returned in the GPR are determined by the function code (FC) specified by the instruction. The name of each function and its associated RTL are shown in Figure 3.
<table>
<thead>
<tr>
<th>Function Code</th>
<th>GPR operands</th>
<th>Storage operands</th>
<th>Function name and RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>RT, RT+1</td>
<td>mem(EA, s)</td>
<td>Fetch and Add</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>RT, RT+1</td>
<td>mem(EA, s)</td>
<td>Fetch and XOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>RT, RT+1</td>
<td>mem(EA, s)</td>
<td>Fetch and OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>RT, RT+1</td>
<td>mem(EA, s)</td>
<td>Fetch and AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>RT, RT+1</td>
<td>mem(EA, s)</td>
<td>Fetch and MaximumUnsigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>RT, RT+1</td>
<td>mem(EA, s)</td>
<td>Fetch and MaximumSigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>RT, RT+1</td>
<td>mem(EA, s)</td>
<td>Fetch and MinimumUnsigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>RT, RT+1</td>
<td>mem(EA, s)</td>
<td>Fetch and MinimumSigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>RT, RT+1</td>
<td>mem(EA, s)</td>
<td>Swap</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>RT, RT1, RT+2</td>
<td>mem(EA, s)</td>
<td>Compare and Swap NotEqual</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>RT</td>
<td>mem(EA, s)</td>
<td>Fetch and IncrementBounded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem(EA+s, s)</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>RT</td>
<td>Function Codes</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>-------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td>RT</td>
<td>mem(EA, s) mem(EA+s, s) Fetch and Increment Equal</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>t ← mem(EA, s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>t2 ← mem(EA+s, s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if t = t2 then</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem(EA, s) ← t+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RT ← t</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else RT ← 1 &lt;&lt; (s*8-1)</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>RT</td>
<td>mem(EA-s, s) mem(EA, s) Fetch and Decrement Bounded</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>t ← mem(EA, s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>t2 ← mem(EA-s, s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if t ≠ t2 then</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem(EA, s) ← t-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RT ← t</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else RT ← 1 &lt;&lt; (s*8-1)</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

s = operand size in number of bytes

Function codes not listed in this table are considered invalid.

For word atomics, only the least significant word of each source register is used, and the least significant word of the target register is updated with the result, while the upper word is set to zero.

**Figure 3.** Load Atomic function codes
Load Word Atomic X-form

\[ \text{lwat} \quad \text{RT,RA,FC} \]

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>f1</th>
<th>16</th>
<th>21</th>
<th>582</th>
<th>31</th>
</tr>
</thead>
</table>

if RA=0 then EA \leftarrow 0 
else \quad EA \leftarrow (RA) 
(RT32:63, \text{mem(EA,4)}) \leftarrow \text{lat}(\text{mem(EA-4,12)}, RT+132:63, RT+232:63, FC) 
RT0:31 \leftarrow 0 

Let the effective address (EA) be (RA). The least significant word of RT and the word of storage at EA are updated as specified by load atomic function code FC. The most significant word of RT is set to zero. Input operands are function code specific, and may include the least significant words of RT+1 and RT+2, and mem(EA-4,12) 

Figure 3 contains the valid function codes. An attempt to execute \textit{lwat} specifying an invalid function code will cause the system data storage error handler to be invoked. 

EA must be a multiple of 4, and the portion of mem(EA-4,12) accessed by the instruction must be contained within an aligned 32-byte block of storage. If either of these requirements is not satisfied, the system alignment error handler is invoked. 

**Special Registers Altered:** 
None

Load Doubleword Atomic X-form

\[ \text{idat} \quad \text{RT,RA,FC} \]

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>f1</th>
<th>16</th>
<th>21</th>
<th>614</th>
<th>31</th>
</tr>
</thead>
</table>

if RA=0 then EA \leftarrow 0 
else \quad EA \leftarrow (RA) 
(RT, \text{mem(EA,8)}) \leftarrow \text{lat}(\text{mem(EA-8,24)}, RT+1, RT+2, FC) 

Let the effective address (EA) be (RA). RT and the doubleword of storage at EA are updated as specified by load atomic function code FC. Input operands are function code specific, and may include RT+1, RT+2, and mem(EA-8,24) 

Figure 3 contains the valid function codes. An attempt to execute \textit{idat} specifying an invalid function code will cause the system data storage error handler to be invoked. 

EA must be a multiple of 8, and the portion of mem(EA-8,24) accessed by the instruction must be contained within an aligned 32-byte block of storage. If either of these requirements is not satisfied, the system alignment error handler is invoked. 

**Special Registers Altered:** 
None
4.5.2 Store Atomic

The Atomic Stores perform an atomic update to an aligned memory location. The manipulation performed on the memory value is determined by the function code (FC) specified by the instruction. The name of each function and its associated RTL are shown in Figure 4.

<table>
<thead>
<tr>
<th>Function Code</th>
<th>GPR operands</th>
<th>Storage operands</th>
<th>Function name and RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>RS</td>
<td>mem(EA,s)</td>
<td>Store Add</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t \leftarrow \text{mem}(EA, s)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t2 \leftarrow t + (RS)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mem(EA,s) $\leftarrow t2$</td>
</tr>
<tr>
<td>00001</td>
<td>RS</td>
<td>mem(EA,s)</td>
<td>Store XOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t \leftarrow \text{mem}(EA, s)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t2 \leftarrow t \oplus (RS)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mem(EA,s) $\leftarrow t2$</td>
</tr>
<tr>
<td>00010</td>
<td>RS</td>
<td>mem(EA,s)</td>
<td>Store OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t \leftarrow \text{mem}(EA, s)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t2 \leftarrow t \mid (RS)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mem(EA,s) $\leftarrow t2$</td>
</tr>
<tr>
<td>00011</td>
<td>RS</td>
<td>mem(EA,s)</td>
<td>Store AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t \leftarrow \text{mem}(EA, s)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t2 \leftarrow t &amp; (RS)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mem(EA,s) $\leftarrow t2$</td>
</tr>
<tr>
<td>00100</td>
<td>RS</td>
<td>mem(EA,s)</td>
<td>Store Maximum Unsigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t \leftarrow \text{mem}(EA, s)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if $(RS) &gt; t$ then mem(EA,s) $\leftarrow (RS)$</td>
</tr>
<tr>
<td>00101</td>
<td>RS</td>
<td>mem(EA,s)</td>
<td>Store Maximum Signed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t \leftarrow \text{mem}(EA, s)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if $(RS) &gt; t$ then mem(EA,s) $\leftarrow (RS)$</td>
</tr>
<tr>
<td>00110</td>
<td>RS</td>
<td>mem(EA,s)</td>
<td>Store Minimum Unsigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t \leftarrow \text{mem}(EA, s)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if $(RS) &lt; t$ then mem(EA,s) $\leftarrow (RS)$</td>
</tr>
<tr>
<td>00111</td>
<td>RS</td>
<td>mem(EA,s)</td>
<td>Store Minimum Signed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t \leftarrow \text{mem}(EA, s)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if $(RS) &lt; t$ then mem(EA,s) $\leftarrow (RS)$</td>
</tr>
<tr>
<td>11000</td>
<td>RS</td>
<td>mem(EA,s) mem(EA+s, s)</td>
<td>Store Twin</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t \leftarrow \text{mem}(EA, s)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$t2 \leftarrow \text{mem}(EA+s, s)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if $t = t2$ then</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mem(EA,s) $\leftarrow (RS)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mem(EA+s,s) $\leftarrow (RS)$</td>
</tr>
</tbody>
</table>

Notes:
- $s$ = operand size in number of bytes
- Function codes not listed in this table are considered invalid.
- For word atomics, only the least significant word of each source register is used.

Figure 4. Store Atomic function codes
Store Word Atomic X-form

stwat RS,RA,FC

if RA=0 then EA ← 0
else EA ← (RA)
mem(EA,8) ← stat(mem(EA,8), RS32:63, FC)

Let the effective address (EA) be (RA). Four or eight bytes of storage at EA are updated as specified by store atomic function code FC. Input operands are function code specific, and may include RS32:63 and mem(EA,8).

Figure 4 contains the valid function codes. An attempt to execute stwat specifying an invalid function code will cause the system data storage error handler to be invoked.

EA must be a multiple of 4, and the portion of mem(EA,8) accessed by the instruction must be contained within an aligned 32-byte block of storage. If either of these requirements is not satisfied, the system alignment error handler is invoked.

Special Registers Altered:
None

Store Doubleword Atomic X-form

stdat RS,RA,FC

if RA=0 then EA ← 0
else EA ← (RA)
mem(EA,16) ← stat(mem(EA,16), RS, FC)

Let the effective address (EA) be (RA). Eight or sixteen bytes of storage at EA are updated as specified by store atomic function code FC. Input operands are function code specific, and may include RS and mem(EA,16).

Figure 4 contains the valid function codes. An attempt to execute stdat specifying an invalid function code will cause the system data storage error handler to be invoked.

EA must be a multiple of 8, and the portion of mem(EA,16) accessed by the instruction must be contained within an aligned 32-byte block of storage. If either of these requirements is not satisfied, the system alignment error handler is invoked.

Special Registers Altered:
None
4.6 Synchronization Instructions

The synchronization instructions are used to ensure that certain instructions have completed before other instructions are initiated, or to control storage access ordering, or to support debug operations.

4.6.1 Instruction Synchronize Instruction

**Instruction Synchronize XL-form**

isync

Executing an *isync* instruction ensures that all instructions preceding the *isync* instruction have completed before the *isync* instruction completes, and that no subsequent instructions are initiated until after the *isync* instruction completes. It also ensures that all instruction cache block invalidations caused by *icbi* instructions preceding the *isync* instruction have been performed with respect to the processor executing the *isync* instruction, and then causes any prefetched instructions to be discarded.

Except as described in the preceding sentence, the *isync* instruction may complete before storage accesses associated with instructions preceding the *isync* instruction have been performed.

This instruction is context synchronizing (see Book III).

**Special Registers Altered:**

None

4.6.2 Load And Reserve and Store Conditional Instructions

The *Load And Reserve and Store Conditional* instructions can be used to construct a sequence of instructions that appears to perform an atomic update operation on an aligned storage location. See Section 1.7.2, “Atomic Update” for additional information about these instructions.

The *Load And Reserve and Store Conditional* instructions are fixed-point *Storage Access* instructions; see Section 3.3.1, “Fixed-Point Storage Access Instructions”, in Book I.

The storage location specified by the *Load And Reserve* and *Store Conditional* instructions must be in storage that is Memory Coherence Required if the location may be modified by another processor or mechanism. If the specified location is in storage that is Write Through Required or Caching Inhibited, the system data storage error handler is invoked.

The *Load And Reserve* instructions include an Exclusive Access hint (EH), which can be used to indicate that the instruction sequence being executed is implementing one of two types of algorithms:

**Atomic Update (EH=0)**

This hint indicates that the program is using a fetch and operate (e.g., fetch and add) or some similar algorithm and that all programs accessing the shared variable are likely to use a similar operation to access the shared variable for some time.

**Exclusive Access (EH=1)**

This hint indicates that the program is attempting to acquire a lock and if it succeeds, will perform another store to the lock variable (releasing the lock) before another program attempts to modify the lock variable.

**Programming Note**

The Memory Coherence Required attribute on other processors and mechanisms ensures that their stores to the reservation granule will cause the reservation created by the *Load And Reserve* instruction to be lost.
**Programming Note**

Because the Load And Reserve and Store Conditional instructions have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, locking, etc.; see Appendix B) that are needed by application programs. Application programs should use these library programs, rather than use the Load And Reserve and Store Conditional instructions directly.

**Programming Note**

EH = 1 should be used when the program is obtaining a lock variable which it will subsequently release before another program attempts to perform a store to it. When contention for a lock is significant, using this hint may reduce the number of times a cache block is transferred between processor caches.

EH = 0 should be used when all accesses to a mutex variable are performed using an instruction sequence with Load And Reserve followed by Store Conditional (e.g., emulating atomic update primitives such as "Fetch and Add;" see Appendix B). The processor may use this hint to optimize the cache to cache transfer of the block containing the mutex variable, thus reducing the latency of performing an operation such as 'Fetch and Add'.

**Programming Note**

Warning: On some processors that comply with versions of the architecture that precede Version 2.00, executing a Load And Reserve instruction in which EH = 1 will cause the illegal instruction error handler to be invoked.

---

**Load Byte And Reserve Indexed X-form**

\[
\text{lbarx RT,RA,RB,EH} \\
\begin{array}{cccccc}
31 & 28 & 24 & 20 & 16 & 12 \\
\text{RT} & \text{RA} & \text{RB} & \text{EA} & \text{EH} \\
0 & 6 & 11 & 16 & 21 & 31
\end{array}
\]

\[
\begin{align*}
\text{if RA} &= 0 \ \text{then b} \leftarrow 0 \\
\text{else} & \quad \text{b} \leftarrow (RA) \\
\text{EA} & \leftarrow (RA)+(RB) \\
\text{RESERVE} & \leftarrow 1 \\
\text{RESERVE\_LENGTH} & \leftarrow 1 \\
\text{RESERVE\_ADDR} & \leftarrow \text{real\_addr}(EA) \\
\text{RT} & \leftarrow 56:63 \ | | \ \text{MEM}(EA, 1)
\end{align*}
\]

Let the effective address (EA) be the sum (RA|0)+(RB).

This instruction creates a reservation for use by a stbcx or waitrsv instruction. A real address computed from the EA as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 1 byte is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the byte in storage addressed by EA before some other processor attempts to modify it.

0 Other programs might attempt to modify the byte in storage addressed by EA regardless of the result of the corresponding stbcx instruction.

1 Other programs will not attempt to modify the byte in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

**Special Registers Altered:**

None

---

**Programming Note**

lbarx serves as both a basic and an extended mnemonic. The Assembler will recognize a lbarx mnemonic with four operands as the basic form, and a lbarx mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.
Load Halfword And Reserve Indexed X-form

**lharx RT,RA,RB,EH**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----|
| 0  | 6  | 11 | 16 | 21 | 31 |

if RA = 0 then b ← 0
else b ← (RA)

EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 2
RESERVE_ADDR ← real_addr(EA)
RT ← 160 || MEM(EA, 2)

Let the effective address (EA) be the sum (RA|0)+(RB).
The halfword in storage addressed by EA is loaded into RT48:63. RT0:47 are set to 0.

This instruction creates a reservation for use by a sthcx. or waitrsv instruction. A real address computed from the EA as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 2 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the halfword in storage addressed by EA before some other processor attempts to modify it.

0 Other programs might attempt to modify the halfword in storage addressed by EA regardless of the result of the corresponding sthcx. instruction.
1 Other programs will not attempt to modify the halfword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 2. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

**Special Registers Altered:**
None

**Programming Note**

*lharx* serves as both a basic and an extended mnemonic. The Assembler will recognize a *lharx* mnemonic with four operands as the basic form, and a *lharx* mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Load Word And Reserve Indexed X-form

**lwarx RT,RA,RB,EH**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----|
| 0  | 6  | 11 | 16 | 21 | 31 |

if RA = 0 then b ← 0
else b ← (RA)

EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 4
RESERVE_ADDR ← real_addr(EA)
RT ← 160 || MEM(EA, 4)

Let the effective address (EA) be the sum (RA|0)+(RB).
The word in storage addressed by EA is loaded into RT32:63. RT0:31 are set to 0.

This instruction creates a reservation for use by a stwcx. or waitrsv instruction. A real address computed from the EA as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 4 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the word in storage addressed by EA before some other processor attempts to modify it.

0 Other programs might attempt to modify the word in storage addressed by EA regardless of the result of the corresponding stwcx. instruction.
1 Other programs will not attempt to modify the word in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

**Special Registers Altered:**
None

**Programming Note**

*lwarx* serves as both a basic and an extended mnemonic. The Assembler will recognize a *lwarx* mnemonic with four operands as the basic form, and a *lwarx* mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.
**Store Byte Conditional Indexed X-form**

stbcx. RS,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>f1</th>
<th>f16</th>
<th>21</th>
<th>694</th>
<th>1</th>
<th>31</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 1 & RESERVE_ADDR = real_addr(EA) then
    MEM(EA, 1) ← (RS)56:63
    undefined_case ← 0
    store_performed ← 1
  else
    z ← smallest real page size supported by implementation
    if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
      undefined_case ← 1
    else
      undefined_case ← 0
      store_performed ← 0
else
  undefined_case ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 1) ← (RS)56:63
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XERSO
else
  CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 1 byte, and the real storage location specified by the **stbcx**. is the same as the real storage location specified by the **lbarx** instruction that established the reservation, (RS)56:63 are stored into the byte in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 1 byte or the real storage location specified by the **stbcx**. is not the same as the real storage location specified by the **lbarx** instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the **stbcx**. is in the same aligned z-byte block of real storage as the real storage location specified by the **lbarx** instruction that established the reservation, it is undefined whether (RS)56:63 are stored into the byte in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

\[ CR0_{LT \; GT \; EQ \; SO} = 0b00 || n || XERSO \]

The reservation is cleared.

**Special Registers Altered:**

CR0
**Store Halfword Conditional Indexed X-form**

sthcx. RS,RA,RB

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>726</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>n1</td>
<td>n6</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
if RESERVE then
if RESERVE_LENGTH = 2 &
RESERVE_ADDR = real_addr(EA) then
MEM(EA, 2) ← (RS)48:63
undefined_case ← 0
store_performed ← 1
else
z ← smallest real page size supported by
implementation
if RESERVE_ADDR = z = real_addr(EA) then
undefined_case ← 1
else
undefined_case ← 0
store_performed ← 0
else
undefined_case ← 0
store_performed ← 0
if undefined_case then
u1 ← undefined 1-bit value
if u1 then
MEM(EA, 2) ← (RS)48:63
u2 ← undefined 1-bit value
CR0 ← 0b00 || u2 || XERSO
else
CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 2 bytes, and the real storage location specified by the **sthcx.** is the same as the real storage location specified by the **lharx** instruction that established the reservation, (RS)48:63 are stored into the halfword in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 2 bytes or the real storage location specified by the **sthcx.** is not the same as the real storage location specified by the **lharx** instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the **sthcx.** is in the same aligned z-byte block of real storage as the real storage location specified by the **lharx** instruction that established the reservation, it is undefined whether (RS)48:63 are stored into the halfword in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

\[ CR_0LT_GT_EQ_S0 = 0b00 \ || \ n \ || XERSO \]

The reservation is cleared.

EA must be a multiple of 2. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

**Special Registers Altered:**

CR0
Store Word Conditional Indexed X-form

```plaintext
stwcx. RS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>15</th>
<th>10</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>RA</td>
<td>RB</td>
<td>150</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

if RA = 0 then b = 0
else b = (RA)
EA = b + (RB)
if RESERVE then
if RESERVE_LENGTH = 4 & RESERVE_ADDR = real_addr(EA) then
MEM(EA, 4) = (RS)32:63
undefined_case = 0
store_performed = 1
else
z = smallest real page size supported by implementation
if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
undefined_case = 1
else
undefined_case = 0
store_performed = 0
else
undefined_case = 0
store_performed = 0
if undefined_case then
u1 = undefined 1-bit value
if u1 then
MEM(EA, 4) = (RS)32:63
u2 = undefined 1-bit value
CR0 = 0b00 || u2 || XER0
else
CR0 = 0b00 || store_performed || XER0
RESERVE = 0

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 4 bytes, and the real storage location specified by the `stwcx.` is the same as the real storage location specified by the `lwarx` instruction that established the reservation, (RS)32:63 are stored into the word in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 4 bytes or the real storage location specified by the `stwcx.` is not the same as the real storage location specified by the `lwarx` instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the `stwcx.` is in the same aligned z-byte block of real storage as the real storage location specified by the `lwarx` instruction that established the reservation, it is undefined whether (RS)32:63 are stored into the word in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

```
CR0LT GT EQ SO = 0b00 || n || XER0
```

The reservation is cleared.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:
CR0
4.6.2.1 64-Bit Load And Reserve and Store Conditional Instructions

**Load Doubleword And Reserve Indexed X-form**

Idarx  
RT,RA,RB,EH

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>84</th>
<th>EH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

If RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
RESERVE ← 1
RESERVE_LENGTH ← 8
RESERVE_ADDR ← real_addr(EA)
RT ← MEM(EA, 8)

Let the effective address (EA) be the sum (RA|0)+ (RB). The doubleword in storage addressed by EA is loaded into RT.

This instruction creates a reservation for use by a `stdcx.` or `waitrsv` instruction. A real address computed from the EA as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 8 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the doubleword in storage addressed by EA before some other processor attempts to modify it.

0 Other programs might attempt to modify the doubleword in storage addressed by EA regardless of the result of the corresponding `stdcx.` instruction.

1 Other programs will not attempt to modify the doubleword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

**Special Registers Altered:**

None

**Programming Note**

Idarx serves as both a basic and an extended mnemonic. The Assembler will recognize a `Idarx` mnemonic with four operands as the basic form, and a `ldarx` mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>214</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

If RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
if RESERVE then

if RESERVE_LENGTH = 8 &
RESERVE_ADDR = real_addr(EA) then
MEM(EA, 8) ← (RS)
undefined_case ← 0
store_performed ← 1
else
z ← smallest real page size supported by implementation
if RESERVE_ADDR × z = real_addr(EA) × z then
undefined_case ← 1
else
undefined_case ← 0
store_performed ← 0
else
undefined_case ← 0
store_performed ← 0
else
undefined_case ← 0
store_performed ← 0
if undefined_case then
u1 ← undefined 1-bit value
if u1 then
MEM(EA, 8) ← (RS)
u2 ← undefined 1-bit value
CR0 ← 0b00 || u2 || XERSO
else
CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 8 bytes, and the real storage location specified by the `stdcx.` is the same as the real storage location specified by the `ldarx` instruction that established the reservation, (RS) is stored into the doubleword in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 8 bytes or the real storage location specified by the `stdcx.` is not the same as the real storage location specified by the `ldarx` instruction that established the reservation, no store is performed.

If a reservation does not exist, no store is performed.

Store Doubleword Conditional Indexed X-form

stdcx.  
RS,RA,RB

---
CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

\[ CR_{0LT} \land GT \land EQ_{SO} = 0b00 \parallel n \parallel XER_{SO} \]

The reservation is cleared.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

**Special Registers Altered:**

CR0
4.6.2.2 128-bit Load And Reserve and Store Conditional Instructions

For \textit{lqarx}, the quadword in storage addressed by \( EA \) is loaded into an even-odd pair of GPRs as follows. In Big-Endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by \( EA \) and the odd-numbered GPR is loaded with the doubleword addressed by \( EA+8 \). In Little-Endian mode, the even-numbered GPR is loaded with the byte-reversed doubleword from storage addressed by \( EA+8 \) and the odd-numbered GPR is loaded with the byte-reversed doubleword addressed by \( EA \).

In the preferred form of the \textit{Load Quadword} instruction \( RA \neq RTp+1 \) and \( RB \neq RTp+1 \).

\textbf{Load Quadword And Reserve Indexed X-form}

\begin{verbatim}
  lqarx RTp,RA,RB,EH
  0  31 R 9 11 16 21 276 31
  0  6 11 16 21 276 31

  if RA = 0 then b \leftarrow 0
  else b \leftarrow (RA)
  EA \leftarrow b + (RB)
  RESERVE \leftarrow 1
  RESERVE_LENGTH \leftarrow 16
  RESERVE_ADDR \leftarrow \text{real_addr}(EA)
  RTp \leftarrow \text{MEM}(EA, 16)
  
  Let the effective address (EA) be the sum \((RA|0)+(RB)\). The quadword in storage addressed by \( EA \) is loaded into \( RTp \).

  This instruction creates a reservation for use by a \textit{stqcx} or \textit{waitrsv} instruction. A real address computed from the \( EA \) as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 16 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

  The value of EH provides a hint as to whether the program will perform a subsequent store to the doubleword in storage addressed by \( EA \) before some other processor attempts to modify it.

  0 Other programs might attempt to modify the doubleword in storage addressed by \( EA \) regardless of the result of the corresponding \textit{stqcx} instruction.

  1 Other programs will not attempt to modify the doubleword in storage addressed by \( EA \) until the program that has acquired the lock performs a subsequent store releasing the lock.

\end{verbatim}

EA must be a multiple of 16. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

If \( RTp \) is odd, \( RTp=RA \), or \( RTp=RB \) the instruction form is invalid. If \( RTp=RA \) or \( RTp=RB \), an attempt to execute this instruction will invoke the system illegal instruction error handler. (The \( RTp=RA \) case includes the case of \( RTp=RA=0 \).)

\textbf{Special Registers Altered:}

None

\begin{boxed-caption}
\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
  & 0 & 6 & 11 & 16 & 21 & 276 & 31 \\
\hline
  lqarx & 5 & 4 & 3 & 2 & 1 & 0 & 31 \\
\hline
\end{tabular}
\end{center}
\end{boxed-caption}

\textit{lqarx} serves as both a basic and an extended mnemonic. The Assembler will recognize a \textit{lqarx} mnemonic with four operands as the basic form, and a \textit{lqarx} mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

\begin{center}
\begin{tabular}{|c|c|}
\hline
  Programming Note & \textit{lqarx} serves as both a basic and an extended mnemonic. The Assembler will recognize a \textit{lqarx} mnemonic with four operands as the basic form, and a \textit{lqarx} mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0. \\
\hline
\end{tabular}
\end{center}
Store Quadword Conditional Indexed X-form

\[ \text{stqcx.} \quad \text{RSp}, \text{RA}, \text{RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RSp</td>
<td>RA</td>
<td>RB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 16 &
    RESERVE_ADDR = real_addr(EA) then
    MEM(EA, 16) ← (RSp)
    undefined_case ← 0
    store_performed ← 1
  else
    z ← smallest real page size supported by implementation
    if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
      undefined_case ← 1
    else
      undefined_case ← 0
      store_performed ← 0
  else
    undefined_case ← 0
    store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 16) ← (RSp)
    u2 ← undefined 1-bit value
    CR0 ← 0b00 || u2 || XER_S0
  else
    CR0 ← 0b00 || store_performed || XER_S0
RESERVE ← 0

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 16 bytes, and the real storage location specified by the \text{stqcx.} is the same as the real storage location specified by the \text{lqarx} instruction that established the reservation, (RSp) is stored into the quadword in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 16 bytes or the real storage location specified by the \text{stqcx.} is not the same as the real storage location specified by the \text{lqarx} instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the \text{stqcx.} is in the same aligned z-byte block of real storage as the real storage location specified by the \text{lqarx} instruction that established the reservation, it is undefined whether (RSp) is stored into the quadword in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

\[ \text{CR0} \Lt \text{GT} \Eq S0 = 0b00 || n || \text{XER_S0} \]

The reservation is cleared.

EA must be a multiple of 16. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

If RSp is odd, the instruction form is invalid.

**Special Registers Altered:**

CR0
4.6.3 Memory Barrier Instructions

The Memory Barrier instructions can be used to control the order in which storage accesses and data transfers are performed. Additional information about these instructions and about related aspects of storage management can be found in Book III.

**Synchronize X-form**

<table>
<thead>
<tr>
<th></th>
<th>L</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if SC¹0 then switch(SC)
  case(1): stncisync
  case(2): stcisync
  case(3): stsync
else switch(L)
  case(0): hwsync
  case(1): lwsync
  case(2): ptesync
  case(4): phwsync
  case(5): plwsync

The *sync* instruction creates a memory barrier (see Section 1.7.1). The set of storage accesses and/or data transfers that is ordered by the memory barrier depends on the contents of the L and SC fields as follows.

- **SC=0**
  - L=0 ("heavyweight sync")
    The memory barrier provides an ordering function for the storage accesses and data transfers associated with all instructions that are executed by the processor executing the *sync* instruction with the exception of *dcbfps* and *dcbstps*. The applicable pairs are all pairs $a_i, b_j$ of such storage accesses and data transfers in which $b_j$ is a data access or data transfer, except that if $a_i$ is the storage access caused by an *icbi* instruction then $b_j$ may be performed with respect to the processor executing the *sync* instruction before $a_i$ is performed with respect to that processor.
  - L=1 ("lightweight sync")
    The memory barrier provides an ordering function for the storage accesses caused by *Load*, *Store*, and *dcbz* instructions that are executed by the processor executing the *sync* instruction and for which the specified storage location is in storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The applicable pairs are all pairs $a_i, b_j$ of such storage accesses except those in which $a_i$ is a storage access caused by a *Store* or *dcbz* instruction and $b_j$ is an access caused by a *Load* instruction.
  - L=2 ("ptesync")
    The set of storage accesses that is ordered by the memory barrier is described in Section 6.9.2 of Book III, as are additional properties of the *sync* instruction with L=2.
  - L=4 ("persistent heavyweight sync")
    The ordering done by the memory barrier is the same as for *sync* with L=0, but extended by adding accesses caused by *dcbfps* and *dcbstps* to both the set A and the set B of the barrier. In addition, the memory barrier ensures that all stores for which the modifications are written to persistent storage by preceding *dcbfps* and *dcbstps* instructions have updated persistent storage before any data access or data transfer caused by subsequent instructions is initiated.
  - L=5 ("persistent lightweight sync")
    The ordering done by the memory barrier is the same as for *sync* with L=1, but extended by adding accesses caused by *dcbfps* and *dcbstps* to both the set A and the set B of the barrier. In addition, the memory barrier ensures that all stores for which the modified...
tions are written to persistent storage by preceding \texttt{dcbfps} and \texttt{dcbstps} instructions have updated persistent storage before any store in set B updates persistent storage.

The ordering done by the memory barrier is cumulative (regardless of the L and SC values).

If L=0 or L=4 (or L=2), the \texttt{sync} instruction has the following additional properties.

- Executing the \texttt{sync} instruction ensures that all instructions preceding the \texttt{sync} instruction have completed before the \texttt{sync} instruction completes, and that no subsequent instructions are initiated until after the \texttt{sync} instruction completes.

- The \texttt{sync} instruction is execution synchronizing (see Book III). However, address translation and reference and change recording (see Book III) associated with subsequent instructions may be performed before the \texttt{sync} instruction completes.

- The memory barrier provides the additional ordering function such that if a given instruction that is the result of a store in set B is executed, all applicable storage accesses in set A have been performed with respect to the processor executing the instruction to the extent required by the associated memory coherence properties. The single exception is that any storage access in set A that is caused by an \texttt{icbi} instruction executed by the processor executing the \texttt{sync} instruction (P1) may not have been performed with respect to P1 (see the description of the \texttt{icbi} instruction on page 1052).

The cumulative properties of the memory barrier apply to the execution of the given instruction as they would to a load that returned a value that was the result of a store in set B.

\begin{center}
\textbf{Programming Note}
\end{center}

Section 1.8 contains a detailed description of how to modify instructions such that a well-defined result is obtained.

The L values 3, 6, and 7 are reserved.

The \texttt{sync} instruction may complete before storage accesses associated with instructions preceding the \texttt{sync} instruction have been performed.

Figure 5 shows the valid combinations of SC and L values. Instructions that use any of these combinations will execute correctly on processors that comply with versions of the architecture that precede Version 3.1 (in which versions the L field is two bits long, the SC field does not exist, and bits 8 and 14:15 of the \texttt{sync} instruction are reserved) and on processors that comply with Version 3.1 and subsequent versions of the architecture. If any other combination is used, the instruction form is invalid.

\begin{center}
\textbf{Figure 5. Interpretation of the L and SC fields}
\end{center}

\begin{table}
\begin{tabular}{|c|c|l|l|}
\hline
SC & L & Intended barrier for processors that comply with V. 3.1 or later & Intended barrier for processors that comply with V. 3.0C or earlier \\
\hline
1 & 1 & stncisync & lw sync \\
2 & 0 & stcisync & hwsync \\
3 & 0 & stsync & hwsync \\
0 & 0 & hwsync & hwsync \\
0 & 1 & lw sync & lw sync \\
0 & 2 & ptesync & ptesync \\
0 & 4 & phwsync & hwsync* \\
0 & 5 & plwsync & lw sync* \\
\hline
\end{tabular}
\end{table}

* depends on details of the bus interface design to have proper persistent storage semantics

\begin{center}
\textbf{Extended: Equivalent to:}
\end{center}

| lw sync | sync 1,0 |
| ptesync | sync 2,0 |
| phwsync | sync 4,0 |
| plwsync | sync 5,0 |
| stncisync | sync 1,1 |
| stcisync | sync 0,2 |
| stsync | sync 0,3 |

Except in the \texttt{sync} instruction description in this section, references to "\texttt{sync}" in Books I-III imply L=0.

\begin{center}
\textbf{Chapter 4. Storage Control Instructions 1087}
\end{center}
unless otherwise stated or obvious from context; the appropriate extended mnemonics are used when other L values are intended. Throughout Books I-III, references to the L field imply SC=0 unless otherwise stated or obvious from context. The SC field is mentioned explicitly, or the appropriate extended mnemonics are used, when non-zero SC values are intended. Some programming examples and recommendations assume a programming model that does not include the store-specific variants of \texttt{sync}. Improved performance may be achieved through the use of store-specific memory barriers in some cases.

---

**Programming Note**

\texttt{sync} serves as both a basic and an extended mnemonic. The Assembler will recognize a \texttt{sync} mnemonic with two operands as the basic form, and a \texttt{sync} mnemonic with one operand or with no operand as an extended form. In the extended form with one operand the SC operand is omitted and assumed to be 0. In the extended form with no operand the L and SC operands are omitted and assumed to be 0.

---

**Programming Note**

The \texttt{sync} instruction can be used to ensure that all stores into a data structure, caused by \texttt{Store} instructions executed in a “critical section” of a program, will be performed with respect to another processor before the store that releases the lock is performed with respect to that processor; see Section B.2, “Lock Acquisition and Release, and Related Techniques” on page 1109.

The memory barrier created by a \texttt{sync} instruction with L=1 (or with SC=0) does not order implicit storage accesses or instruction fetches. The memory barrier created by a \texttt{sync} instruction with L=0 (or L=2) orders implicit storage accesses and instruction fetches associated with instructions preceding the \texttt{sync} instruction but not those associated with instructions following the \texttt{sync} instruction.

In order to obtain the best performance across the widest range of implementations, the programmer should use the \texttt{sync} instruction with L=1 or with SC=0, or the \texttt{eieio} instruction, if any of these is sufficient for his needs; otherwise he should use \texttt{sync} with L=0 (or with L=4 or L=5 if he needs to order accesses to persistent storage). \texttt{sync} with L=2 should not be used by application programs.

---

**Programming Note**

The functions provided by \texttt{sync} with L=1 and with SC=0 are a strict subset of those provided by \texttt{sync} with L=0. (The functions provided by \texttt{sync} with L=2 are a strict superset of those provided by \texttt{sync} with L=4; see Book III.)

---

**Enforce In-order Execution of I/O X-form**

\texttt{eieio}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
31 & 30 & 29 & 28 & 27 & 854\
\hline
\end{tabular}

The \texttt{eieio} instruction creates a memory barrier (see Section 1.7.1, “Storage Access Ordering”), which provides an ordering function for the storage accesses caused by \texttt{Load}, \texttt{Store}, and \texttt{dcbz} instructions executed by the processor executing the \texttt{eieio} instruction. These storage accesses are divided into the two sets listed below. The storage access caused by a \texttt{dcbz} instruction is ordered as a store.

1. Loads and stores to storage that is both Caching Inhibited and Guarded, and stores to main storage caused by stores to storage that is Write Through Required.

   The applicable pairs are all pairs $a_i, b_j$ of such accesses.

2. Stores to storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited.

   The applicable pairs are all pairs $a_i, b_j$ of such accesses.

The operations caused by the stream variants of the \texttt{dcbt} and \texttt{dcbtst} instructions (i.e., the providing of hints) are ordered by \texttt{eieio} as a third set of operations, the operations caused by \texttt{tlbie} and \texttt{tlbsync} instructions (see Book III) are ordered by \texttt{eieio} as a fourth set of operations, and the operations caused by \texttt{slbieg} or \texttt{slbias} and \texttt{slbsync} instructions (see Book III) are ordered by \texttt{eieio} as a fifth set of operations.

Each of the five sets of storage accesses or operations is ordered independently of the other four sets. The ordering done by \texttt{eieio}'s memory barrier for the second set is cumulative; the ordering done by \texttt{eieio}'s memory barrier for the other four sets is not cumulative.

The \texttt{eieio} instruction may complete before storage accesses associated with instructions preceding the \texttt{eieio} instruction have been performed. The \texttt{eieio} instruction may complete before operations caused by \texttt{dcbt} and \texttt{dcbtst} instructions preceding the \texttt{eieio} instruction have been performed.

**Special Registers Altered:**

None
The **eieio** instruction is intended for use in doing memory-mapped I/O. Because loads, and separately stores, to storage that is both Caching Inhibited and Guarded are performed in program order (see Section 1.7.1, “Storage Access Ordering” on page 1034), **eieio** is needed for such storage only when loads must be ordered with respect to stores.

For the **eieio** instruction, accesses in set 1, \(a_i\) and \(b_j\) need not be the same kind of access or be to storage having the same storage control attributes. For example, \(a_i\) can be a load to Caching Inhibited, Guarded storage, and \(b_j\) a store to Write Through Required storage.

If stronger ordering is desired than that provided by **eieio**, the **sync** instruction must be used, with the appropriate value in the L field.

---

**Programming Note**

The functions provided by **eieio** for its second set are a strict subset of those provided by **sync** with \(L=1\).
4.6.4 Wait Instruction

The wait instruction is used to stop instruction fetching and execution until certain events occur. These events include exceptions (see Section 1.2.1 of Book III), event-based branch exceptions (see Section 1.1), the passage of a specified amount of time, and the modification of a storage location.

**Wait X-form**

```
wait WC,PL
```

The `wait` instruction causes instruction fetching and execution to be suspended under certain conditions, depending on the values of the WC and PL fields. Instruction fetching and execution are resumed when the events specified by the WC field occur or in the rare case of an implementation-dependent event.

The values of the WC field are as follows.

<table>
<thead>
<tr>
<th>WC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Resume instruction fetching and execution when an exception or event-based branch exception occurs.</td>
</tr>
<tr>
<td>1</td>
<td>Resume instruction fetching and execution when an exception or event-based branch exception occurs, or when a reservation made by the processor does not exist (see Section 1.7.2.1).</td>
</tr>
<tr>
<td>2</td>
<td>Resume instruction fetching and execution when an exception or event-based branch exception occurs, or when the amount of time specified by the PL field has passed.</td>
</tr>
<tr>
<td>3</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

The values of the PL field are as follows.

<table>
<thead>
<tr>
<th>PL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>A short wait time is specified.</td>
</tr>
<tr>
<td>11</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

If WC=0, or if WC=1 and a reservation made by the processor exists, or if WC=2 and a value for PL that is not reserved is specified, the following applies.

- Upon completion of the instruction, instruction fetching and execution is suspended.
- Instruction fetching and execution resumes when any of the following conditions are met.
  - An exception or event-based branch exception occurs.
  - WC=1 and a reservation made by the processor does not exist.
  - WC=2 and the specified amount of time has passed.
  - An implementation-dependent event occurs.

---

**Programming Note**

Because the waiting begins when the instruction completes, if the waiting is ended by an exception that causes a change of control flow (interrupt, event-based branch), the SPR that is set to reflect the point in the instruction stream at which the change of control flow occurred (e.g., SRR0 for a Decrementer interrupt) will contain the EA of the instruction following the wait instruction.

If WC=1 and a reservation made by the processor does not exist, or if WC=2 and a reserved value of PL is specified, the instruction is treated as a no-op.

**Programming Note**

Bits 6 and 7 of the `wait` instruction may be used in some implementations for an implementation-dependent field. Unless the intention is to use the implementation-dependent field, these bits must be coded zero.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Examples of extended mnemonics for `wait`:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait</td>
<td>wait 0,0</td>
</tr>
<tr>
<td>wait 0</td>
<td>wait 0,0</td>
</tr>
<tr>
<td>wait 1</td>
<td>wait 1,0</td>
</tr>
<tr>
<td>waitcsv</td>
<td>wait 1,0</td>
</tr>
<tr>
<td>pause_short</td>
<td>wait 2,0</td>
</tr>
</tbody>
</table>

Except in this section, references to “`wait`” in Books I-III include all defined forms of `wait` unless otherwise stated or obvious from context.

**Programming Note**

`wait` serves as both a basic and an extended mnemonic. The Assembler will recognize a `wait` mnemonic with two operands as the basic form and a `wait` mnemonic with one operand or with no operand as an extended form. In the extended form with one operand the PL operand is omitted and assumed to be 0. In the extended form with no operand the WC and PL operands are omitted and assumed to be 0.
The `wait` instruction frees computational resources which might be allocated to another program or converted into power savings.

Since exceptions corresponding to system-caused interrupts (see Section 7.4 of Book III) may occur at any time, including immediately prior to the `wait` instruction, applications should not depend on them to cause `wait` to resume. In order to ensure timely resumption, therefore, applications should execute `wait` only in order to suspend processing until an event-based branch exception or loss of reservation occurs or a specified amount of time has passed.

Also, since exceptions corresponding to interrupts can cause `wait` to resume at any time without any EBB exception or loss of reservation having occurred, and in consideration of the possibility of resuming because of an implementation-dependent event, programs that execute `wait` should check that the expected condition has actually occurred after the `wait` instruction completes. If the expected condition has not occurred, `wait` should be re-executed. An example code usage is shown below.

```c
while (!expected_condition), wait
```

Applications that execute `wait` in order to suspend processing until an external event-based branch exception occurs (see Section 6.2) should enable external event-based branch exceptions (by setting BESCREE=1) and disable event-based branches (by setting BESCRGE=0) before executing `wait`. If BESCRGE=1, then the expected event-based branch exception may cause the corresponding event-based branch to occur immediately prior to execution of the `wait` instruction. This will result in a hang condition since the EBB exception that was expected to cause `wait` to resume will have already occurred.
Chapter 5. Time Base

The Time Base (TB) is a 64-bit register (see Figure 6) containing a 64-bit unsigned integer that is incremented periodically as described below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBU</td>
<td>Upper 32 bits of Time Base</td>
</tr>
<tr>
<td>TBL</td>
<td>Lower 32 bits of Time Base</td>
</tr>
</tbody>
</table>

Figure 6. Time Base

The Time Base monotonically increments until its value becomes \( 0xFFFF_FFFF_FFFF_FFFF \) (\( 2^{64} - 1 \)); at the next increment its value becomes \( 0x0000_0000_0000_0000 \). There is no interrupt or other indication when this occurs.

The suggested frequency at which the time base increments is 512 MHz, however, variation from this rate is allowed provided the following requirements are met.

- The contents of the Time Base differ by no more than \( +/- \) four counts from what they would be if they incremented at the required frequency.
- Bit 63 of the Time Base is set to 1 between 30% and 70% of the time over any time interval of at least 16 counts.

The Power ISA does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock. The Time Base update frequency is not required to be constant. What is required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base is under the control of the system software.

Programming Note

If the operating system initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from \( 2^{64} - 1 \) to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.
5.1 Time Base Instructions

**Move From Time Base XFX-form**

mftb RT, TBR

[Phased-Out]

This instruction behaves as if it were an mfspr instruction; see the mfspr instruction description in Section 3.3.18 of Book I.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Extended mnemonics for Move From Time Base:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mftb Rx</td>
<td>mftb Rx, 268</td>
</tr>
<tr>
<td></td>
<td>mfspr Rx, 268</td>
</tr>
<tr>
<td>mftbu Rx</td>
<td>mftbRx, 269</td>
</tr>
<tr>
<td></td>
<td>mfsprRx, 269</td>
</tr>
</tbody>
</table>

**Programming Note**

New programs should use mfspr instead of mftb to access the Time Base.

**Programming Note**

mftb serves as both a basic and an extended mnemonic. The Assembler will recognize an mftb mnemonic with two operands as the basic form, and an mftb mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB).
Programming Note

Since the update frequency of the Time Base is implementation-dependent, the algorithm for converting the current value in the Time Base to time of day is also implementation-dependent.

As an example, assume that the Time Base increments at the constant rate of 512 MHz. (Note, however, that programs should allow for the possibility that some implementations may not increment the least-significant 4 bits of the Time Base at a constant rate.) What is wanted is the pair of 32-bit values comprising a POSIX standard clock: the number of whole seconds that have passed since 00:00:00 January 1, 1970, UTC, and the remaining fraction of a second expressed as a number of nanoseconds.

Assume that:

- The value 0 in the Time Base represents the start time of the POSIX clock (if this is not true, a simple 64-bit subtraction will make it so).
- The integer constant \( \text{ticks}_{\text{per sec}} \) contains the value 512,000,000, which is the number of times the Time Base is updated each second.
- The integer constant \( \text{ns}_{\text{adj}} \) contains the value \( \frac{1,000,000,000}{512,000,000} \times 2^{32}/2 = 4194304000 \)
  which is the number of nanoseconds per tick of the Time Base, multiplied by \( 2^{32} \) for use in \( \text{mulhwu} \) (see below), and then divided by 2 in order to fit, as an unsigned integer, into 32 bits.

When the processor is in 64-bit mode, The POSIX clock can be computed with an instruction sequence such as this:

```assembly
mfspr Ry,268  # Ry = Time Base
lwz Rx,\text{ticks}_{\text{per sec}}
divdu Rz,Ry,Rx  # Rz = whole seconds
stw Rz,\text{posix sec}
mulld Rz,Rz,Rx  # Rz = quotient * divisor
sub Rz,Ry,Rz  # Rz = excess ticks
lwz Rx,\text{ns}_{\text{adj}}
slw1 Rz,Rz,1  # Rz = 2 * excess ticks
mulhwu Rz,Rz,Rx  # mul by (ns/tick)/2 * 2^{32}
stw Rz,\text{posix ns}# product[0:31] = excess ns
```

Non-constant update frequency

In a system in which the update frequency of the Time Base may change over time, it is not possible to convert an isolated Time Base value into time of day. Instead, a Time Base value has meaning only with respect to the current update frequency and the time of day that the update frequency was last changed. Each time the update frequency changes, either the system software is notified of the change via an interrupt (see Book III), or the change was instigated by the system software itself. At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of \( \text{ticks}_{\text{per sec}} \) for the new frequency, and save the time of day, Time Base value, and tick rate. Subsequent calls to compute Time of Day use the current Time Base Value and the saved value.

---

Chapter 6. Event-Based Branch Facility

6.1 Event-Based Branch Overview

The Event-Based Branch facility allows application programs to enable hardware to change the effective address of the next instruction to be executed when certain events occur to an effective address specified by the program.

The operation of the Event-Based Branch facility is summarized as follows:

- The Event-Based Branch facility is available only when the system software has made it available. See Section 10.5 of Book III for additional information.
- When the Event-Based Branch facility is available, event-based branches are caused by event-based exceptions. Event-based exceptions can be enabled to occur by setting bits in the BESCR.
- When an event-based exception occurs, the bit in the BESCR control field corresponding to the event-based exception is set to 0 and the bit in the Event Status field in the BESCR corresponding to the event-based exception is set to 1.
- If the global enable bit in the BESCR is set to 1 when any of the bits in the status field are set to 1 (i.e., when an event-based exception exists), an event-based branch occurs.
- The event-based branch causes the following to occur.
  - The global enable bit is set to 0.
  - Bits 0:61 of the EBBRR are set to the effective address of the instruction that would have attempted to execute next if the event-based branch did not occur.
  - Instruction fetch and execution continues at the effective address contained in the EBBHR.
- The event-based branch handler performs the necessary processing in response to the event, and then executes an `rfebb` instruction in order to resume execution at the instruction at the address indicated in the EBBRR.

Additional information about the Event-Based Branch facility is given in Section 4.4 of Book III.

6.2 Event-Based Branch Registers

6.2.1 Branch Event Status and Control Register

The Branch Event Status and Control Register (BESCR) is a 64-bit register that contains control and status information about the Event-Based Branch facility.
System software controls whether or not event-based branches occur regardless of the contents of the BESCR. See Section 10.4.4 of Book III and Section 7.2.13 of Book III.

The entire BESCR can be read or written using SPR 806. Individual bits of the BESCR can be set or reset using two sets of additional SPR numbers.

- When `mtspr` indicates SPR 800 (Branch Event Status and Control Set, or BESCRS), the bits in BESCR which correspond to "1" bits in the source register are set to 1; all other bits in the BESCR are unaffected. SPR 801 (BESCRSU) provides the same capability to each of the upper 32 bits of the BESCR.

- When `mtspr` indicates SPR 802 (Branch Event Status and Control Reset, or BESCRR), the bits in BESCR which correspond to "1" bits in the source register are set to 0; all other bits in the BESCR are unaffected. SPR 803 (BESCRRU) provides the same capability to each of the upper 32 bits of the BESCR.

When `mfspr` indicates any of the above SPR numbers, the current value of the register is returned.

### Programming Note

Event-based branch handlers typically reset event status bits upon entry, and enable event enable bits after processing an event. Execution of `rfebb` then re-enables the GE bit so that additional event-based branches can occur.

### Figure 7. Branch Event Status and Control Register (BESCR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>GE</td>
<td>Event Control</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 8. Branch Event Status and Control Register Upper (BESCRU)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>GE</td>
<td>Event Control</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>EE</td>
<td>External Event-Based Exception Enable (EE)</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>External event-based (EBB) exceptions are disabled.</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>External EBB exceptions are enabled until an external event-based exception occurs, at which time:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- EE is set to 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- EEO is set to 1</td>
</tr>
</tbody>
</table>

External event-based exceptions exist in any privilege state when an external EBB input from the platform is active. See the system documentation for information about the external EBB input.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>PME</td>
<td>Performance Monitor Event-Based Exception Enable (PME)</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Performance Monitor event-based exceptions are disabled.</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Performance Monitor event-based exceptions are enabled until a Performance Monitor event-based exception occurs, at which time:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- PME is set to 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- PMEO is set to 1</td>
</tr>
</tbody>
</table>

See Chapter 10 of Book III for information about Performance Monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

### Programming Note

Performance Monitor event-based exceptions can only occur in problem state. See Section 10.2 of Book III.

### Reserved

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:33</td>
<td></td>
<td>Bits 32:33 must contain 0b00 when <code>rfebb</code> is executed; otherwise the instruction is treated as if the instruction form were invalid.</td>
</tr>
</tbody>
</table>

### Event Status

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34:61</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>EEO</td>
<td>External Event-Based Exception Occurred (EEO)</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>An external EBB exception has not occurred since the last time software set this bit to 0.</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>An external EBB exception has occurred since the last time software set this bit to 0.</td>
</tr>
</tbody>
</table>
63 Performance Monitor Event-Based Exception Occurred (PMEO)
0 A Performance Monitor event-based exception has not occurred since the last time software set this bit to 0.
1 A Performance Monitor event-based exception has occurred since the last time software set this bit to 0.

This bit is set to 1 by the hardware when a Performance Monitor event-based exception occurs. This bit can be set to 0 only by the mtspr instruction.

See Chapter 10 of Book III for information about Performance Monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

--- Programming Note ---
As part of processing an External EBB exception, it may also be necessary to perform additional operations to manage the external EBB input from the system. See the system documentation for details.

6.2.2 Event-Based Branch Handler Register

The Event-Based Branch Handler Register (EBBHR) is a 64-bit register that contains the 62 most significant bits of the effective address of the instruction that is executed next after an event-based branch occurs. Bits 62:63 must be available to be read and written by software.

--- Programming Note ---
After handling an event-based branch, software should set the "exception occurred" bit(s) corresponding to the event-based exception(s) that have occurred to 0. See the Programming Notes in Section 6.3 for additional information.

6.2.3 Event-Based Branch Return Register

The Event-Based Branch Return Register (EBBRR) is a 64-bit register that contains the 62 most significant bits of an instruction effective address as specified below.

```
0 Effective Address
62 63
```

--- Figure 9. Event-Based Branch Handler Register (EBBHR) ---

--- Figure 10. Event-Based Branch Return Register (EBBRR) ---

When an event-based branch occurs, bits 0:61 of the EBBRR are set to the effective address of the instruction that would have attempted to execute next if the event-based branch did not occur.

Bits 62:63 are reserved.
6.3 Event-Based Branch Instructions

Return from Event-Based Branch XL-form

rfebb S

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>S</th>
<th>146</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>///</td>
<td>///</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BESCRGE ← S
NIA ← EBBR0:61 || 0b00

BESCRGE is set to S.

If there are no pending event-based exceptions, then the next instruction is fetched from the address EBBR0:61 || 0b00 (when MSR SF=1) or 320 || EBBR32:61 || 0b00 (when MSR SF=0). If one or more pending event-based exceptions exist, an event-based branch is generated; in this case the value placed into EBBR by the Event-Based Branch facility is the address of the instruction that would have been executed next had the event-based branch not occurred.

If BESCR32:33 ≠ 0b00 the instruction is treated as if the instruction form were invalid.

See Section 4.4 of Book III for additional information about this instruction.

Special Registers Altered:

BESCR
MSR (See Book III)

Extended Mnemonics:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfebb</td>
<td>rfebb 1</td>
</tr>
</tbody>
</table>

**Programming Note**

*rfebb* serves as both a basic and an extended mnemonic. The Assembler will recognize an *rfebb* mnemonic with one operand as the basic form, and an *rfebb* mnemonic with no operand as the extended form. In the extended form, the S operand is omitted and assumed to be 1.

---

**Programming Note**

When an event-based branch occurs, the event-based branch handler can execute the following sequence of operations. This sequence of operations assumes that the handler routine has access to a stack or other area in memory in which state information from the main program can be stored. Note also that in this example, the handler entry point is labeled “E,” r1 and r2 are used as scratch registers, and both external EBB and Performance Monitor EBB exceptions are enabled.

E: Save state
- // This is the entry point
- mfspr r1, BESCR // Check event status
- if r163=1, then
  - Process PM exception
  - r2 ← 0x0000 0000 0000 0001
  - mtspr BESCR, r2 // Reset PMEO status bit
  - r2 ← 0x0000 0001 0000 0000
  - mtspr BESCRS, r1 // Re-enable PM exceptions
  // Note: The PMAE bit of MMCR0 must also be enabled. See Book III.
- if r162=1, then
  - Process external exception
  - r2 ← 0x0000 0000 0000 0002
  - mtspr BESCR, r2 // Reset EEO status bit
  - r2 ← 0x0000 0002 0000 0000
  // De-activate external EBB input from platform
  - mtspr BESCRS, r1 // Re-enable external EBB exceptions
  // . . .
  // Other exceptions
  // are processed similarly.
  // . . .
  - Restore state
  - rfebb 1 // return & global enable

Note that before resetting the BESCR_EEO, the external EBB input from the platform should be deactivated, and additional operations to manage the external EBB input may be required. See the system documentation for details.

In the above sequence, if other exceptions occur after they are enabled, another event-based branch will occur immediately after *rfebb* is executed.
Chapter 7. Branch History Rolling Buffer

The Branch History Rolling Buffer (BHRB) is a buffer containing an implementation-dependent number of entries, referred to as BHRB Entries (BHRBEs), that contain information related to branches that have been taken. Entries are numbered from 0 through n, where n is implementation-dependent but no more than 1023. Entry 0 is the most-recently written entry. The BHRB is read by means of the `mfbhrbe` instruction.

System software typically controls the availability of the BHRB as well as the number of entries that it contains. If the BHRB is accessed when it is unavailable, the system facility unavailable error handler is invoked.

Various events or actions by the system software may result in the BHRB occasionally being cleared. If BHRB entries are read after this has occurred, 0s will be returned. See the description of the `mfbhrbe` instruction for additional information.

The BHRB is typically used in conjunction with Performance Monitor event-based branches. (See Chapter 6 of Book II.) When used in conjunction with this facility, BESCR_PME is set to 1 to enable Performance Monitor event-based exceptions, and Performance Monitor alerts are enabled to enable the writing of BHRB entries. When a Performance Monitor alert occurs, Performance Monitor alerts are disabled, BHRB entries are no longer written, and an event-based branch occurs. (See Chapter 10 of Book III for additional information on the Performance Monitor.) The event-based branch handler can then access the contents of the BHRB for analysis.

When the BHRB is written by hardware, only those `Branch` instructions that meet the filtering criteria are written. See Section 10.4.7 of Book III.

The following paragraphs describe the entries written into the BHRB for various types of `Branch` instructions for which the branch was taken. In some circumstances, however, the hardware may be unable to make the entry even though the following paragraphs require it. In such cases, the hardware sets the EA field to 0, and indicates any missed entries using the T and P fields. (See Section 7.1.)

When an I-form or B-form `Branch` instruction is entered into the BHRB, bits 0:61 of the effective address of the `Branch` instruction are written into the next available entry, except that the entry may or may not be written in the following cases.

- The effective address of the branch target exceeds the effective address of the `Branch` instruction by 4.
- The instruction is a B-form `Branch`, the effective address of the branch target exceeds the effective address of the `Branch` instruction by 8, and the instruction immediately following the `Branch` instruction is not another `Branch` instruction.

The determination of whether the effective address of the branch target exceeds the effective address of the `Branch` instruction by 4 or 8 is made modulo $2^{64}$.

Programming Note

The cases described above, for which the BHRBE need not be written, are cases for which some implementations may optimize the execution of the `Branch` instruction (first case) or of the `Branch` instruction and the following instruction (second case) in a manner that makes writing the BHRBE difficult. Such implementations may provide a means by which system software can disable these optimizations, thereby ensuring that the corresponding BHRBEs are written normally.

When an XL-form `Branch` instruction is entered into the BHRB, bits 0:61 of the effective address of the `Branch` instruction are written into the next available entry if allowed by the filtering mode; subsequently, bits 0:61 of the effective address of the branch target are written into the following entry.
7.1 Branch History Rolling Buffer Entry Format

Branch History Rolling Buffer Entries (BHRBEs) have the following format.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>This entry either is not implemented or has been cleared. There are no valid entries beyond the current entry.</td>
</tr>
<tr>
<td>01-11</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

When the EA field contains a zero value, bits 62:63 specify the type of marker as described below.

**Effective Address (EA)**

- When this field is set to a non-zero value, it contains bits 0:61 of the effective address of the instruction indicated by the T field; otherwise this field indicates that the entry is a marker with the meaning specified by the T and P fields.

**Target Address (T)**

- The EA field contains bits 0:61 of the effective address of a Branch instruction for which the branch was taken.
- The EA field contains bits 0:61 of the branch effective address of the branch target of an XL-form Branch instruction for which the branch was taken.

**Prediction (P)**

- When T=0, this field has the following meaning.
  - 0 The outcome of the Branch instruction was correctly predicted.
  - 1 The outcome of the Branch instruction was mispredicted.
- When T=1, this field has the following meaning.
  - 0 The Branch instruction was predicted to be taken and the target address was predicted correctly, or the target address was not predicted because the branch was predicted to be not taken.
  - 1 The target address was mispredicted.

**Programming Note**

It is expected that programs will not contain Branch instructions with instruction or target effective address equal to 0. If such instructions exist, programs cannot distinguish between entries that are markers and entries that correspond to instructions with instruction or target effective address 0.
7.2 Branch History Rolling Buffer Instructions

The Branch History Rolling Buffer instructions enable application programs to clear and read the BHRB. The availability of these instructions is controlled by the system software. (See Chapter 10 of Book III.) When an attempt is made to execute these instructions when they are unavailable, the system facility unavailable error handler is invoked.

Clear BHRB X-form

cirbhrb

for n = 0 to (number_of_BHRBES implemented - 1)
   BHRB(n) ← 0

All BHRB entries are set to 0s.

Special Registers Altered:
   None.

Move From Branch History Rolling Buffer Entry XFX-form

mfbhrbe RT,BHRBE

n ← BHRBE[0:9]
If n < number of BHRBES implemented then
   RT ← BHRBE(n)
else
   RT ← 640

The BHRBE field denotes an entry in the BHRB. If the designated entry is within the range of BHRB entries implemented and Performance Monitor alters are disable (see Section 10.5 of Book III), the contents of the designated BHRB entry are placed into register RT; otherwise, 640s are placed into register RT.

In order to ensure that the current BHRB contents are read by this instruction, one of the following must have occurred prior to this instruction and after all previous Branch and cirbhrb instructions have completed.
   ■ an event-based branch has occurred
   ■ an rfebb (see Chapter 6 of Book II) has been executed
   ■ a context synchronizing event (see Section 1.5 of Book III) other than isync (see Section 4.6.1 of Book II) has occurred.

Special Registers Altered:
   None

---

### Programming Note

In order to read all the BHRB entries containing information about taken branches, software should read the entries starting from entry number 0 and continuing until an entry containing all 0s is read or until all implemented BHRB entries have been read.

Since the number of BHRB entries may decrease or the BHRB may be cleared at any time, if a given entry, m, is read as not containing all 0s and is read again subsequently, the subsequent read may return all 0s even though the program has not executed cirbhrb.
Appendix A. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided for certain instructions. This appendix defines extended mnemonics and symbols related to instructions defined in Book II. Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

A.1 Data Cache Block Touch [for Store] Mnemonics

The TH field in the Data Cache Block Touch and Data Cache Block Touch for Store instructions control the actions performed by the instructions. Extended mnemonics are provided that represent the TH value in the mnemonic rather than requiring it to be coded as a numeric operand.

- \text{dcbtct RA, RB, TH} \quad (\text{equivalent to: dcbt for TH values of } 0b00000 \text{ - } 0b00111); \quad \text{other TH values are invalid.}

- \text{dcbtds RA, RB, TH} \quad (\text{equivalent to: dcbt for TH values of } 0b00000 \text{ or } 0b01000 \text{ - } 0b01111); \quad \text{other TH values are invalid.}

- \text{dcbtt RA, RB} \quad (\text{equivalent to: dcbt for TH value of } 0b10000)

- \text{dcbna RA, RB} \quad (\text{equivalent to: dcbt for TH value of } 0b10001)

- \text{dcbtstct RA, RB, TH} \quad (\text{equivalent to: dcbtst for TH values of } 0b00000 \text{ or } 0b00000 \text{ - } 0b00111); \quad \text{other TH values are invalid.}

- \text{dcbtstds RA, RB, TH} \quad (\text{equivalent to: dcbtst for TH values of } 0b00000 \text{ or } 0b01000 \text{ - } 0b01111); \quad \text{other TH values are invalid.}

- \text{dcbtstt RA, RB} \quad (\text{equivalent to: dcbtst for TH value of } 0b10000)

A.2 Data Cache Block Flush Mnemonics

The L field in the Data Cache Block Flush instruction controls the scope of the flush function performed by the instruction, or the scope of the store function when \( L=6 \). Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

- \text{dcbf RA, RB} \quad (\text{equivalent to: dcbf RA, RB, 0})

- \text{dcbfl RA, RB} \quad (\text{equivalent to: dcbf RA, RB, 1})

- \text{dcbflp RA, RB} \quad (\text{equivalent to: dcbf RA, RB, 4})

- \text{dcbfps RA, RB} \quad (\text{equivalent to: dcbf RA, RB, 6})

A.3 Or Mnemonics

The three register fields in the or instruction can be used to specify a hint indicating how the processor should handle stores caused by previous Store or dcbz instructions. An extended mnemonic is supported that represents the operand values in the mnemonic rather than requiring them to be coded as numeric operands.

- \text{miso} \quad (\text{equivalent to: or } 26,26,26)

A.4 Load And Reserve Mnemonics

The EH field in the Load And Reserve instructions provides a hint regarding the type of algorithm implemented by the instruction sequence being executed. Extended mnemonics are provided that allow the EH value to be omitted and assumed to be 0b0.

- \text{ibarx, ibarx, iwarx, idarx, and iqarx} serve as both basic and extended mnemonics. The Assembler will recognize these mnemonics with four operands as the basic form, and these mnemonics with three oper-
ands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

\[
\begin{align*}
\text{lbarx} & \quad \text{RT,RA,RB} \quad (\text{equivalent to: } \text{lbarx} \quad \text{RT,RA,RB,0}) \\
\text{lharc} & \quad \text{RT,RA,RB} \quad (\text{equivalent to: } \text{lharc} \quad \text{RT,RA,RB,0}) \\
\text{lwarx} & \quad \text{RT,RA,RB} \quad (\text{equivalent to: } \text{lwarx} \quad \text{RT,RA,RB,0}) \\
\text{ldarx} & \quad \text{RT,RA,RB} \quad (\text{equivalent to: } \text{ldarx} \quad \text{RT,RA,RB,0}) \\
\text{lqarx} & \quad \text{RT,RA,RB} \quad (\text{equivalent to: } \text{lqarx} \quad \text{RT,RA,RB,0}) \\
\text{sync} & \quad (\text{equivalent to: } \text{sync} \quad 0,0) \\
\text{sync} \ x & \quad (\text{equivalent to: } \text{sync} \quad x,0) \\
\text{hwsync} & \quad (\text{equivalent to: } \text{sync} \quad 0,0) \\
\text{lwsync} & \quad (\text{equivalent to: } \text{sync} \quad 1,0) \\
\text{ptesync} & \quad (\text{equivalent to: } \text{sync} \quad 2,0) \\
\text{phwsync} & \quad (\text{equivalent to: } \text{sync} \quad 4,0) \\
\text{plwsync} & \quad (\text{equivalent to: } \text{sync} \quad 5,0) \\
\text{stncisync} & \quad (\text{equivalent to: } \text{sync} \quad 1,1) \\
\text{stcisync} & \quad (\text{equivalent to: } \text{sync} \quad 0,2) \\
\text{stsync} & \quad (\text{equivalent to: } \text{sync} \quad 0,3) \\
\text{wait} & \quad (\text{equivalent to: } \text{wait} \quad 0,0) \\
\text{wait} \ rsrv & \quad (\text{equivalent to: } \text{wait} \quad 1,0) \\
\text{pause} \_short & \quad (\text{equivalent to: } \text{wait} \quad 2,0) \\
\text{mftb} \ Rx & \quad (\text{equivalent to: } \text{mftb} \ Rx,268) \\
& \quad \text{or: } \text{mfspr} \ Rx,268 \\
\text{mftbu} \ Rx & \quad (\text{equivalent to: } \text{mftb} \ Rx,269) \\
& \quad \text{or: } \text{mfspr} \ Rx,269 \\
\text{rfebb} & \quad (\text{equivalent to: } \text{rfebb} \ 1)
\end{align*}
\]

### A.5 Synchronize Mnemonics

The L and SC fields in the **Synchronize** instruction control the scope of the synchronization function performed by the instruction. Extended mnemonics are provided that represent the L and SC values in the mnemonic rather than requiring them to be coded as numeric operands.

**Note:** **sync** serves as both a basic and an extended mnemonic. The Assembler will recognize a **sync** mnemonic with two operands as the basic form, and a **sync** mnemonic with one operand or with no operand as an extended form. In the extended form with one operand the SC operand is omitted and assumed to be 0. In the extended form with no operand the L and SC operands are omitted and assumed to be 0.

### A.6 Wait Mnemonics

The WC field in the **wait** instruction determines the conditions under which instruction execution resumes. Extended mnemonics are provided that represent the WC and PL values in the mnemonic rather than requiring them to be coded as numeric operands.

**Note:** **wait** serves as both a basic and an extended mnemonic. The Assembler will recognize a **wait** mnemonic with two operands as the basic form and a **wait** mnemonic with one operand or with no operand as an extended form. In the extended form with one operand the PL operand is omitted and assumed to be 0. In the extended form with no operand the WC and PL operands are omitted and assumed to be 0.

### A.7 Move To/From Time Base Mnemonics

The tbr field in the **Move From Time Base** instruction specifies whether the instruction reads the entire Time Base or only the high-order half of the Time Base.

- mftb Rx \quad (equivalent to: mftb Rx,268)
- mftbu Rx \quad (equivalent to: mftb Rx,269)

### A.8 Return From Event-Based Branch Mnemonic

The S field in the **Return from Event-Based Branch** instruction specifies the value to which the instruction sets the GE field in the BESCR. Extended mnemonics are provided that represent the S value in the mnemonic rather than requiring it to be coded as a numeric operand.

- rfebb \quad (equivalent to: rfebb \ 1)

**Note:** **rfebb** serves as both a basic and an extended mnemonic. The Assembler will recognize this mnemonic with one operand as the basic form, and this mnemonic with no operands as the extended form. In the extended form the S operand is omitted and assumed to be 1.
Appendix B. Programming Examples for Sharing Storage

This appendix gives examples of how dependencies and the Synchronization instructions can be used to control storage access ordering when storage is shared between programs.

Many of the examples use extended mnemonics (e.g., bne, bne-, cmpw) that are defined in Appendix C of Book I.

Many of the examples use the Load And Reserve and Store Conditional instructions, in a sequence that begins with a Load And Reserve instruction and ends with a Store Conditional instruction (specifying the same storage location as the Load Conditional) followed by a Branch Conditional instruction that tests whether the Store Conditional instruction succeeded.

In these examples it is assumed that contention for the shared resource is low; the conditional branches are optimized for this case by using “+” and “-” suffixes appropriately.

The examples deal with words; they can be used for doublewords by changing all word-specific mnemonics to the corresponding doubleword-specific mnemonics (e.g., lwarx to ldarx, cmpw to cmpd).

In this appendix it is assumed that all shared storage locations are in storage that is Memory Coherence Required, and that the storage locations specified by Load And Reserve and Store Conditional instructions are in storage that is neither Write Through Required nor Caching Inhibited.

B.1 Atomic Update Primitives

This section gives examples of how the Load And Reserve and Store Conditional instructions can be used to emulate atomic read/modify/write operations.

An atomic read/modify/write operation reads a storage location and writes its next value, which may be a function of its current value, all as a single atomic operation. The examples shown provide the effect of an atomic read/modify/write operation, but use several instructions rather than a single atomic instruction.

Fetch and No-op

The “Fetch and No-op” primitive atomically loads the current value in a word in storage.

In this example it is assumed that the address of the word to be loaded is in GPR 3 and the data loaded are returned in GPR 4.

```
loop:
  lwax r4,0,r3 #load and reserve
  stwcx. r4,0,r3 #store old value if
    # still reserved
  bne- loop #loop if lost reservation
```

Note:

1. The stwcx, if it succeeds, stores to the target location the same value that was loaded by the preceding lwax. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the lwax is still the current value at the time the stwcx is executed.

Fetch and Store

The “Fetch and Store” primitive atomically loads and replaces a word in storage.

In this example it is assumed that the address of the word to be loaded and replaced is in GPR 3, the new value is in GPR 4, and the old value is returned in GPR 5.

```
loop:
  lwax r5,0,r3 #load and reserve
  stwcx. r4,0,r3 #store new value if
    # still reserved
  bne- loop #loop if lost reservation
```
Fetch and Add

The “Fetch and Add” primitive atomically increments a word in storage.

In this example it is assumed that the address of the word to be incremented is in GPR 3, the increment is in GPR 4, and the old value is returned in GPR 5.

```
loop:
  lwarx r5,0,r3 #load and reserve
  add r0,r4,r5 #increment word
  stwcx. r0,0,r3 #store new value if still res’ved
  bne- loop   #loop if lost reservation
```

Fetch and AND

The “Fetch and AND” primitive atomically ANDs a value into a word in storage.

In this example it is assumed that the address of the word to be ANDed is in GPR 3, the value to AND into it is in GPR 4, and the old value is returned in GPR 5.

```
loop:
  lwarx r5,0,r3 #load and reserve
  and r0,r4,r5 #AND word
  stwcx. r0,0,r3 #store new value if still res’ved
  bne- loop   #loop if lost reservation
```

Note:
1. The sequence given above can be changed to perform another Boolean operation atomically on a word in storage, simply by changing the `and` instruction to the desired Boolean instruction (`or`, `xor`, etc.).

Test and Set

This version of the “Test and Set” primitive atomically loads a word from storage, sets the word in storage to a nonzero value if the value loaded is zero, and sets the EQ bit of CR Field 0 to indicate whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR 3, the new value (nonzero) is in GPR 4, and the old value is returned in GPR 5.

```
loop:
  lwarx r5,0,r3 #load and reserve
  cmpwi r5,0 #done if word not equal to 0
  bne- exit   #skip if not
  stwcx. r5,0,r3 #store new value if still res’ved
  bne- loop   #loop if lost reservation
  mr r4,r6 #return value from storage
```

Compare and Swap

The “Compare and Swap” primitive atomically compares a value in a register with a word in storage, if they are equal stores the value from a second register into the word in storage, if they are unequal loads the word from storage into the first register, and sets the EQ bit of CR Field 0 to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR 3, the comparand is in GPR 4 and the old value is returned there, and the new value is in GPR 5.

```
loop:
  lwarx r6,0,r3 #load and reserve
  cmpw r4,r6 #1st 2 operands equal?
  bne- exit   #skip if not
  stwcx. r5,0,r3 #store new value if still res’ved
  bne- loop   #loop if lost reservation
exit:
  mr r4,r6   #return value from storage
```

Notes:
1. The semantics given for “Compare and Swap” above are based on those of the IBM System/370 Compare and Swap instruction. Other architectures may define a Compare and Swap instruction differently.

2. “Compare and Swap” is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by `lwarx` and `stwcx`. A major weakness of a System/370-style Compare and Swap instruction is that, although the instruction itself is atomic, it checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The sequence shown above has the same weakness.

3. In some applications the second `bne` instruction and/or the `mr` instruction can be omitted. The `bne`- is needed only if the application requires that if the EQ bit of CR Field 0 on exit indicates “not equal” then (r4) and (r6) are in fact not equal. The `mr` is needed only if the application requires that if the comparands are not equal then the word from storage is loaded into the register with which it was compared (rather than into a third register). If either or both of these instructions is omitted, the resulting Compare and Swap does not obey System/370 semantics.
B.2 Lock Acquisition and Release, and Related Techniques

This section gives examples of how dependencies and the \textit{Synchronization} instructions can be used to implement locks, import and export barriers, and similar constructs.

B.2.1 Lock Acquisition and Import Barriers

An “import barrier” is an instruction or sequence of instructions that prevents storage accesses caused by instructions following the barrier from being performed before storage accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. A \textit{sync} instruction can be used as an import barrier, but the approaches shown below will generally yield better performance because they order only the relevant storage accesses.

B.2.1.1 Acquire Lock and Import Shared Storage

If \textit{lwarx} and \textit{stwcx} instructions are used to obtain the lock, an import barrier can be constructed by placing an \textit{isync} instruction immediately following the loop containing the \textit{lwarx} and \textit{stwcx}. The following example uses the “Compare and Swap” primitive to acquire the lock.

In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the old value of the pointer is returned in GPR 5. The value to be added to the pointer is in GPR 6, and the address of the shared data structure is in GPR 9.

\begin{verbatim}
loop:
lwarx r6,0,r3,1 #load lock and reserve
cmpw r4,r6    #skip ahead if
bne- wait    # lock not free
stwcx. r5,0,r3 #try to set lock
bne- loop    #loop if lost reservation
isync        #import barrier
lwz r7,data1(r9)#load shared data
.
.
wait...     #wait for lock to free

The hint provided with \textit{lwarx} indicates that after the program acquires the lock variable (i.e., \textit{stwcx} is successful), it will release it (i.e., store to it) prior to another program attempting to modify it.

The second \textit{bne-} does not complete until CR0 has been set by the \textit{stwcx}. The \textit{stwcx} does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the \textit{stwcx} completes successfully. Together, the second \textit{bne-} and the subsequent \textit{lwz}do not need \textit{isync} because the load uses the \textit{stwcx} that created the reservation used by the successful \textit{stwcx}. If the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, an \textit{lwsync} instruction can be used instead of the \textit{isync} instruction. If \textit{lwsync} is used, the load from “data1” may be performed before the \textit{stwcx}. But if the \textit{stwcx} fails, the branch is taken and the \textit{lwarx} is re-executed. If the \textit{stwcx} succeeds, the value returned by the load from “data1” is valid even if the load is performed before the \textit{stwcx}, because the \textit{lwsync} ensures that the load is performed after the instance of the \textit{lwarx} that created the reservation used by the successful \textit{stwcx}.

B.2.1.2 Obtain Pointer and Import Shared Storage

If \textit{lwarx} and \textit{stwcx} instructions are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer. The following example uses the “Fetch and Add” primitive to obtain and increment the pointer.

In this example it is assumed that the address of the pointer is in GPR 3, the value to be added to the pointer is in GPR 4, and the old value of the pointer is returned in GPR 5.

\begin{verbatim}
loop:
lwarx r5,0,r3 #load pointer and reserve
add r0,r4,r5 #increment the pointer
stwcx. r0,0,r3 #try to store new value
bne- loop    #loop if lost reservation
lwz r7,data1(r5) #load shared data

The load from “data1” cannot be performed until the pointer value has been loaded into GPR 5 by the \textit{lwarx}. The load from “data1” may be performed before the \textit{stwcx}. But if the \textit{stwcx} fails, the branch is taken and the value returned by the load from “data1” is discarded. If the \textit{stwcx} succeeds, the value returned by the load from “data1” is valid even if the load is performed before the \textit{stwcx}, because the load uses the pointer value returned by the instance of the \textit{lwarx} that created the reservation used by the successful \textit{stwcx}.

An \textit{isync} instruction could be placed between the \textit{bne-} and the subsequent \textit{lwz}, but no \textit{isync} is needed if all accesses to the shared data structure depend on the value returned by the \textit{lwarx}.\end{verbatim}
B.2.2 Lock Release and Export Barriers

An “export barrier” is an instruction or sequence of instructions that prevents the store that releases a lock from being performed before stores caused by instructions preceding the barrier have been performed. An export barrier can be used to ensure that all stores to a shared data structure protected by a lock will be performed with respect to any other processor before the store that releases the lock is performed with respect to that processor.

B.2.2.1 Export Shared Storage and Release Lock

A `sync` instruction can be used as an export barrier independent of the storage control attributes (e.g., presence or absence of the Caching Inhibited attribute) of the storage containing the shared data structure. Because the lock must be in storage that is neither Write Through Required nor Caching Inhibited, if the shared data structure is in storage that is Write Through Required or Caching Inhibited a `sync` instruction must be used as the export barrier.

In this example it is assumed that the shared data structure is in storage that is Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw r7,data1(r9)#store shared data (last)
sync #export barrier
stw r4,lock(r3)#release lock
```

The `sync` ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the `sync` have been performed with respect to that processor.

B.2.2.2 Export Shared Storage and Release Lock using `lwsync`

If the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, an `lwsync` instruction can be used as the export barrier. Using `lwsync` rather than `sync` will yield better performance in most systems.

In this example it is assumed that the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw r7,data1(r9)#store shared data (last)
lwsync #export barrier
stw r4,lock(r3)#release lock
```

The `lwsync` ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the `lwsync` have been performed with respect to that processor.

B.2.3 Safe Fetch

If a load must be performed before a subsequent store (e.g., the store that releases a lock protecting a shared data structure), a technique similar to the following can be used.

In this example it is assumed that the address of the storage operand to be loaded is in GPR 3, the contents of the storage operand are returned in GPR 4, and the address of the storage operand to be stored is in GPR 5.

```
lwz r4,0(r3)#load shared data
cmpw r4,r4 #set CR0 to "equal"
bne- $-8 #branch never taken
stw r7,0(r5)#store other shared data
```

An alternative is to use a technique similar to that described in Section B.2.1.2, by causing the `stw` to depend on the value returned by the `lwz` and omitting the `cmpw` and `bne-`. The dependency could be created by ANDing the value returned by the `lwz` with zero and then adding the result to the value to be stored by the `stw`. If both storage operands are in storage that is neither Write Through Required nor Caching Inhibited, another alternative is to replace the `cmpw` and `bne-` with an `lwsync` instruction.

```
lwz r4,0(r3)#load shared data
lwsync #export barrier
stw r7,0(r5)#store other shared data
```
B.3 List Insertion

This section shows how the \texttt{lwarx} and \texttt{stwcx} instructions can be used to implement simple insertion into a singly linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below and requires a more complicated strategy such as using locks.)

The “next element pointer” from the list element after which the new element is to be inserted, here called the “parent element”, is stored into the new element, so that the new element points to the next element in the list; this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR 3, the address of the new element is in GPR 4, and the next element pointer is at offset 0 from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements.

```
loop:
lwarx r2,0,r3 #get next pointer
stw r2,0(r4)#store in new element
lwsync or sync #order stw before stwcx
stwcx. r4,0,r3 #add new element to list
bne- loop #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, “livelock” can occur. (Livelock is a state in which processors interact in a way such that no processor makes forward progress.)

If it is not possible to allocate list elements such that each element’s next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, sequence.

```
lwz r2,0(r3)#get next pointer
loop1:
  mr r5,r2 #keep a copy
  stw r2,0(r4)#store in new element
  sync #order stw before stwcx.
  and before lwarx

loop2:
lwarx r2,0,r3 #get it again
  cmpw r2,r5 #loop if changed (someone
  bne- loop1 # else progressed)
  stwcx. r4,0,r3 #add new element to list
  bne- loop2 #loop if failed
```

In the preceding example, livelock is avoided by the fact that each processor re-executes the \texttt{stw} only if some other processor has made forward progress.

B.4 Notes

The following notes apply to Section B.1 through Section B.3.

1. To increase the likelihood that forward progress is made, it is important that looping on \texttt{lwarx/stwcx} pairs be minimized. For example, in the “Test and Set” sequence shown in Section B.1, this is achieved by testing the old value before attempting the store; were the order reversed, more \texttt{stwcx} instructions might be executed, and reservations might more often be lost between the \texttt{lwarx} and the \texttt{stwcx}.

2. The manner in which \texttt{lwarx} and \texttt{stwcx} are communicated to other processors and mechanisms, and between levels of the storage hierarchy within a given processor, is implementation-dependent. In some implementations performance may be improved by minimizing looping on a \texttt{lwarx} instruction that fails to return a desired value. For example, in the “Test and Set” sequence shown in Section B.1, if the programmer wishes to stay in the loop until the word loaded is zero, he could change the “bne- exit” to “bne- loop”. However, in some implementations better performance may be obtained by using an ordinary Load instruction to do the initial checking of the value, as follows.

```
loop:
lwz r5,0(r3)#load the word
cmpwi r5,0 #loop back if word
bne- loop #  not equal to 0
lwarx r5,0,r3 #try again, reserving
cmpwi r5,0 # (likely to succeed)
  bne- loop
stwcx.r4,0,r3 #try to store non-0
bne- loop #loop if lost reserv'n
```

3. In a multiprocessor, livelock is possible if there is a Store instruction (or any other instruction that can clear another processor’s reservation; see Section 1.7.2.1) between the \texttt{lwarx} and the \texttt{stwcx} of a \texttt{lwarx/stwcx} loop and any byte of the storage location specified by the Store is in the reservation granule. For example, the first code sequence shown in Section B.3 can cause livelock if two list elements have next element pointers in the same reservation granule.
Book III:

Power ISA Operating Environment Architecture
Chapter 1. Introduction

1.1 Overview

Chapter 1 of Book I describes computation modes, document conventions, a general systems overview, instruction formats, and storage addressing. This chapter augments that description as necessary for the Power ISA Operating Environment Architecture.

1.2 Document Conventions

The notation and terminology used in Book I apply to this Book also, with the following substitutions.

- For “system alignment error handler” substitute “Alignment interrupt”.
- For “system data storage error handler” substitute “Data Storage interrupt”, “Hypervisor Data Storage interrupt”, or “Data Segment interrupt”, as appropriate.
- For “system error handler” substitute “interrupt”.
- For “system floating-point enabled exception error handler” substitute “Floating-Point Enabled Exception type Program interrupt”.
- For “system illegal instruction error handler” substitute “Hypervisor Emulation Assistance interrupt”.
- For “system instruction storage error handler” substitute “Instruction Storage interrupt”, “Hypervisor Instruction Storage interrupt”, or “Instruction Segment interrupt”, as appropriate.
- For “system privileged instruction error handler” substitute “Privileged Instruction type Program interrupt”.
- For “system service program” substitute “System Call interrupt” or “System Call Vectored interrupt”, as appropriate.
- For “system trap handler” substitute “Trap type Program interrupt”.
- For “system facility unavailable error handler” substitute “Facility Unavailable interrupt” or “Hypervisor Facility Unavailable interrupt.”

1.2.1 Definitions and Notation

The definitions and notation given in Book I and Book II are augmented by the following.

- **Threaded processor, single-threaded processor, thread**
  A threaded processor implements one or more “threads”, where a thread corresponds to the Book I/II concept of “processor”. That is, the definition of “thread” is the same as the Book I definition of “processor”, and “processor” as used in Books I and II can be thought of as either a single-threaded processor or as one thread of a multi-threaded processor. Except where the meaning is clear in context or the number of threads does not matter, the only unqualified uses of “processor” in Book III are in resource names (e.g. Processor Identification Register); such uses should be regarded as meaning “threaded processor”. The threads of a multi-threaded processor typically share certain resources, such as the hardware components that execute certain kinds of instructions (e.g., Fixed-Point instructions), certain caches, the address translation mechanism, and certain hypervisor and ultravisor resources.

- **real page**
  A unit of real storage that is aligned at a boundary that is a multiple of its size. The real page size is 4KB.

- **context of a program**
  The state (e.g., privilege and relocation) in which the program executes. The context is controlled by the contents of certain System Registers, such as the MSR and PTCR, of certain looksie buffers, such as the SLB and TLB, and of the Page Table.

- **performed**
  The definition of “performed” given in Section 1.1 of Book II is extended to apply to implicit storage accesses and to invalidations of entries in caches of information derived from address translation tables, as follows.
  - The definition of “load is performed” applies to accesses for performing address translation.
- The definition of “store is performed” applies to accesses for recording reference and change information.
- A TLB entry invalidation by thread T1 is performed with respect to thread T2 when the instruction that requested the invalidation has caused the specified entry, if present, to be made invalid in T2’s TLB, and similarly for invalidations of entries in other caches of information derived from tables used in address translation.

**exception**
An error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.

**interrupt**
The act of changing the machine state in response to an exception, as described in Chapter 7. “Interrupts” on page 1247.

- ultravisor interrupt
  An interrupt that forces the thread into ultravisor state by explicitly setting MSR$_SOHV$PR to 0b110 (see Section 4.2.1).
- hypervisor interrupt
  An interrupt that forces the thread into hypervisor state by explicitly setting MSR$_HV$PR to 0b10 and is not an ultravisor interrupt.
  All interrupts explicitly set MSR$_PR$ to 0.

**trap interrupt**
An interrupt that results from execution of a Trap instruction.

**“must”**
If software that runs in hypervisor state violates a rule that is stated using the word “must” (e.g., “this field must be set to 0”), and the rule pertains to the contents of a hypervisor resource, to executing an instruction that can be executed only in hypervisor state, or to accessing storage in real addressing mode, the results are undefined, and may include altering resources belonging to other partitions, causing the system to “hang”, etc. The same is true for software that runs in ultravisor state and violates a “must” rule pertaining to an ultravisor resource or instruction.

**hardware**
Any combination of hard-wired implementation, emulation assist, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of emulation assists or interrupts to implement the architecture is implementation-dependent.

**ultravisor privileged**
A term used to describe an instruction or facility that is available when and only when the thread is in ultravisor state.

**hypervisor privileged**
A term used to describe an instruction or facility that is available when and only when the thread is in hypervisor state.

---

**Programming Note**
Because ultravisor state is also a hypervisor state, hypervisor privileged instructions and facilities are also available when the thread is in ultravisor state. (The distinct privilege states in which a hypervisor privileged instruction or facility is available are: hypervisor non-ultravisor state, and ultravisor state.)

**privileged**
A term used to describe an instruction or facility that is available when and only when the thread is in privileged state.

---

**Programming Note**
Because hypervisor state is also a privileged state, privileged instructions and facilities are also available when the thread is in hypervisor state (and when the thread is in ultravisor state). (The distinct privilege states in which a privileged instruction or facility is available are: privileged non-hypervisor state, hypervisor non-ultravisor state, and ultravisor state.)

**privileged state and supervisor mode**
Used interchangeably to refer to a state in which privileged facilities are available.

**problem state and user mode**
Used interchangeably to refer to a state in which privileged facilities are not available.

$\|^, \|, \||, ...$ denotes a field that is reserved in an instruction, in a register, or in an architected storage table.

$\?, \??, \???,$ ... denotes a field that is implementation-dependent in an instruction, in a register, or in an architected storage table.

1.2.2 Reserved Fields
Book I’s description of the handling of reserved bits in System Registers, and of reserved values of defined fields of System Registers, applies also to the SLB. Book I’s description of the handling of reserved values of defined fields of System Registers applies also to architected storage tables (e.g., the Page Table).
Software should set reserved fields in the SLB and in architected storage tables to zero, because these fields may be assigned a meaning in some future version of the architecture.

Some fields of certain architected storage tables may be written to automatically by the hardware, e.g., Reference and Change bits in the Page Table. When the hardware writes to such a table, the following rules are obeyed.

- Unless otherwise stated, no defined field other than the one(s) specifically being updated are modified.
- Contents of reserved fields are either preserved or written as zero.

### 1.2.3 Deviations from the Sequential Execution Model

Additional exceptions to the rule that the thread obeys the sequential execution model, beyond those described in Section 2.2 of Book I and in the bullet defining “program order” in Section 1.1 of Book II, are the following.

- A System Reset or asynchronous Machine Check interrupt may occur. The determination of whether an instruction is required by the sequential execution model is not affected by the potential occurrence of a System Reset or asynchronous Machine Check interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)

- A context-altering instruction is executed (Chapter 12, “Synchronization Requirements for Context Alterations” on page 1333). The context alteration need not take effect until the required subsequent synchronizing operation has occurred.

- A Reference and Change bit is updated by the thread. The update need not be performed with respect to that thread until the required subsequent synchronizing operation has occurred.

- A Branch instruction is executed and the branch is taken. The update of the Come-From Address Register (see Section 9.2 of Book III) need not occur until a subsequent context synchronizing operation has occurred.

### 1.2.4 Restricting Out-of-Order Execution

Because some classes of security exploits use side-effects of out-of-order execution to infer behavior of or receive information from programs, it may sometimes be necessary to limit out-of-order execution beyond what's necessary to maintain the appearance of compliance with the sequential execution model. This may include restrictions on the otherwise-permitted deviations from the sequential execution model described in Section 1.2.3 and Section 6.5. The Or Immediate instruction described in Section 5.4.3 can be used to create a barrier to out-of-order execution.

### 1.3 General Systems Overview

The hardware contains the sequencing and processing controls for instruction fetch, instruction execution, and interrupt action. Most implementations also contain data and instruction caches. Instructions that the processing unit can execute fall into the following classes:

- instructions executed in the Branch Facility
- instructions executed in the Fixed-Point Facility
- instructions executed in the Floating-Point Facility
- instructions executed in the Vector Facility

Almost all instructions executed in the Branch Facility, Fixed-Point Facility, Floating-Point Facility, and Vector Facility are nonprivileged and are described in Book I. Book II may describe additional nonprivileged instructions (e.g., Book II describes some nonprivileged instructions for cache management). Instructions related to the privileged state, control of hardware resources, control of the storage hierarchy, and all other privileged instructions are described here or are implementation-dependent.

### 1.4 Exceptions

The following augments the exceptions defined in Book I that can be caused directly by the execution of an instruction:

- the execution of a floating-point instruction when MSR_PP=0 (Floating-Point Unavailable interrupt)
- an attempt to modify a hypervisor resource when the thread is in privileged but non-hypervisor state (see Chapter 2), or an attempt to execute a hypervisor-only instruction (e.g., `tlbie`) when the thread is in privileged but non-hypervisor state
- an attempt to modify an ultravisor resource when the thread is in privileged but non-ultravisor state (see Chapter 3), or an attempt to execute an ultravisor-only instruction (e.g., `urfd`, `msgsndu`, `msgclru`) when the thread is in privileged but non-ultravisor state
- the execution of a traced instruction (Trace interrupt)
- the execution of a Vector instruction when the vector facility is unavailable (Vector Unavailable interrupt)
1.5 Synchronization

The synchronization described in this section refers to the state of the thread that is performing the synchronization.

1.5.1 Context Synchronization

An instruction or event is context synchronizing if it satisfies the requirements listed below. Such instructions and events are collectively called context synchronizing operations. The context synchronizing operations are the isync instruction, the System Linkage instructions, the mtmsr[d] instructions with L=0, and most interrupts (see Section 7.4).

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetching mechanism to any instruction execution mechanism) to be halted.

2. The operation is not initiated or, in the case of isync, does not complete, until all instructions that precede the operation have completed to a point at which they have reported all exceptions they will cause.

3. The operation ensures that the instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated, except that the operation has no effect on the context in which the associated Reference and Change bit updates are performed.

4. If the operation directly causes an interrupt (e.g., sc directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt (see Section 7.9).

5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described in Section 6.5, “Performing Operations Out-of-Order”.)

Programming Note

A context synchronizing operation is necessarily execution synchronizing; see Section 1.5.2.

Unlike the Synchronize instruction, a context synchronizing operation does not affect the order in which storage accesses are performed.

Item 2 permits a choice only for isync (and [p]hwsync and ptesync; see Section 1.5.2) because all other execution synchronizing operations also alter context.
1.5.2 Execution Synchronization

An instruction is \textit{execution synchronizing} if it satisfies items 2 and 3 of the definition of context synchronization (see Section 1.5.1). [\texttt{phw}sync and \texttt{ptesync} are treated like \texttt{isync} with respect to item 2. The execution synchronizing instructions are \texttt{phw}sync, \texttt{ptesync}, the \texttt{mtmsr[d]} instructions with \( L = 1 \), and all context synchronizing instructions.

\begin{center}
\textbf{Programming Note}

Unlike a context synchronizing operation, an execution synchronizing instruction does not ensure that the instructions following that instruction will execute in the context established by that instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context synchronizing operation.
\end{center}
Chapter 2. Logical Partitioning (LPAR) and Thread Control

2.1 Overview

The Logical Partitioning (LPAR) facility permits threads and portions of real storage to be assigned to logical collections called partitions, such that a program executing on a thread in one partition cannot interfere with any program executing on a thread in a different partition. This isolation can be provided for both problem state and privileged non-hypervisor state programs, by using a layer of trusted software, called a hypervisor program (or simply a "hypervisor"), and the resources provided by this facility to manage system resources. (A hypervisor is a program that runs in hypervisor state; see below.)

The number of partitions supported is implementation-dependent.

A thread is assigned to one partition at any given time. A thread can be assigned to any given partition without consideration of the physical configuration of the system (e.g., shared registers, caches, organization of the storage hierarchy), except that threads that share certain hypervisor resources may need to be assigned to the same partition; see Section 2.6. The registers and facilities used to control Logical Partitioning are listed below and described in the following subsections.

Except in the following subsections, references to the "operating system" in this document include the hypervisor unless otherwise stated or obvious from context.

2.2 Logical Partitioning Control Register (LPCR)

The contents of the LPCR control a number of aspects of the operation of the thread with respect to a logical partition. Below are shown the bit definitions for the LPCR.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:3</td>
<td>Virtualization Control (VC)</td>
</tr>
<tr>
<td></td>
<td>Controls the virtualization of partition memory for partitions that use HPT translation. This field contains three subfields, VPM, ISL, and KBV. Accesses that are initiated in hypervisor state (i.e., MSR_hv PR=0b10) are performed as if VPM=0 and KBV=0. (ISL applies regardless of privilege.)</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Virtualized Partition Memory (VPM)</td>
</tr>
<tr>
<td></td>
<td>Controls whether VPM mode is enabled when address translation is enabled as specified below.</td>
</tr>
<tr>
<td></td>
<td>0 - VPM mode disabled</td>
</tr>
<tr>
<td></td>
<td>1 - VPM mode enabled</td>
</tr>
<tr>
<td></td>
<td>When address translation is disabled, VPM mode is enabled. See Section 6.7.2, &quot;Virtualized Partition Memory (VPM) Mode&quot;, and Section 6.7.3.3, &quot;Virtual Real Mode Addressing Mechanism&quot;, for additional information on VPM mode.</td>
</tr>
</tbody>
</table>

**Programming Note**

VPM must be set to zero by hypervisors that use HPT translation and want to receive storage interrupts from applications running directly under them as DSIs and ISIs (instead of HDSIs and HISIs).

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Ignore SLB Large Page Specification (ISL)</td>
</tr>
<tr>
<td></td>
<td>Controls whether ISL mode is enabled as specified below.</td>
</tr>
<tr>
<td></td>
<td>0 - ISL mode disabled</td>
</tr>
<tr>
<td></td>
<td>1 - ISL mode enabled</td>
</tr>
<tr>
<td></td>
<td>When ISL mode is enabled and address translation is enabled, address translation is performed as if the contents of SLB_{LLL} and PRTE_{STPS} were 0b000. When address translation is disabled, the setting of the ISL bit has no effect. ISL mode has no effect on SLB, TLB, and ERAT entry invalidations.</td>
</tr>
</tbody>
</table>
caused by \textit{slbie}, \textit{slbieg}, \textit{slbia}, \textit{slbiag}, \textit{tlbie}, and \textit{tlbiel}.

\textbf{Programming Note}:
Specifying that L|LP = 0b000 in \text{PATE\textsubscript{PS}} has the same effect on address translation when translation is disabled as enabling ISL mode when translation is enabled.

ISL mode is needed when a partition is running with address translation enabled because translation uses the SLB, and the contents of the SLB are controlled by the operating system and should not be modified by the hypervisor. ISL mode is not needed when a partition is running with address translation disabled since Virtual Real Mode address translation uses \text{PATE\textsubscript{PS}}, which is not visible to the operating system and is in complete control of the hypervisor.

Any PTE with a base page size of 4K satisfies the page size portion of the match criteria for HPT search.

\textbf{3 Key-Based Virtualization (KBV)}
Controls whether Key-Based Virtualization is enabled as specified below.

0 - KBV is disabled
1 - KBV is enabled

When KBV is enabled and MSR\textsubscript{HV|PR} = 0b10, Virtual Page Class Key Storage Protection exceptions that occur on storage operand accesses when VPM = 0 cause Hypervisor Data Storage interrupts.

\textbf{Programming Note}:
Key-Based Virtualization provides an efficient means for the hypervisor to intercept storage references, e.g. MMIO, that must be emulated. (The corresponding behavior for instruction fetching is not desired.) Virtual Page Class Key Storage Protection exceptions not handled by the hypervisor should be reflected to the operating system at its Data Storage interrupt vector with the hypervisor having set DSISR\textsubscript{42}.

\textbf{9:11 Default Prefetch Depth (DPFD)}
The DPFD field is used as the default prefetch depth for data stream prefetching when DSCR\textsubscript{DPFD} = 0; see page 1054.

\textbf{12:16 Reserved}

\textbf{17:19 Power-saving mode Exit Cause Enable (Upper Section) (PECE\textsubscript{U})}

\textbf{17 Hypervisor Virtualization Exit Enable}
0 When the \textit{stop} instruction is executed with PSSCR\textsubscript{EC} = 1, Hypervisor Virtualization exceptions are not enabled to cause exit from power-saving mode.
1 When the \textit{stop} instruction is executed with PSSCR\textsubscript{EC} = 1, Hypervisor Virtualization exceptions are enabled to cause exit from power-saving mode.

\textbf{18:19 Reserved}

\textbf{37 Hypervisor Alternate Interrupt Location (HAIL)}
Controls the effective address offset, or alternate effective address for System Call Vectored, of the interrupt handler and the relocation mode in which it begins execution for all interrupts received in hypervisor state except those subject to the overrides described below.

0 The interrupt is taken with MSR\textsubscript{IR DR} = 0b00 and no effective address offset or alternate effective address.
1 The interrupt is taken with MSR\textsubscript{IR DR} = 0b11. If the interrupt is not System Call Vectored, an effective address offset of 0xc000_0000_0000_4000 is applied. System Call Vectored uses an alternate effective address of 0xc000_0000_0000_3||LEV||0b0_0000.

The overrides mentioned above are as follows. The list should be read from the top down; the first item matching a given situation applies.
- If the interrupt is received in ultrvisor state, the interrupt is taken as if LPCR\textsubscript{HAIL} = 0.
- Machine Check, System Reset, and Hypervisor Maintenance interrupts are taken as if LPCR\textsubscript{HAIL} = 0.
- If the interrupt occurs when MSR\textsubscript{HV} = 1 and either MSR\textsubscript{IR} = 0 or MSR\textsubscript{DR} = 0, the interrupt is taken as if LPCR\textsubscript{HAIL} = 0.

\textbf{38 Interrupt Little-Endian (ILE)}
The contents of the ILE bit are copied into MSR\textsubscript{LE} by interrupts that set MSR\textsubscript{HV} to 0 (see
Section 7.5), to establish the Endian mode for the interrupt handler.

**Alternate Interrupt Location (AIL)**

Controls the effective address offset, or alternate effective address for System Call Vectored, of the interrupt handler and the relocation mode in which it begins execution for all interrupts received in privileged non-hypervisor state except those subject to the overrides described below.

- **0**: The interrupt is taken with MSR\textsubscript{IR} \textsubscript{DR} = 0b00 and no effective address offset or alternate effective address.
- **1**: Reserved
- **2**: Reserved
- **3**: The interrupt is taken with MSR\textsubscript{IR} \textsubscript{DR} = 0b11. If the interrupt is not System Cal Vectored, an effective address offset of 0xc00_0000_0000_4000 is applied. System Call Vectored uses an alternate effective address of 0xc00_0000_0000_3 || LEV || 0b0_0000.

The overrides mentioned above are as follows. The list should be read from the top down; the first item matching a given situation applies.

- If the interrupt occurs when MSR\textsubscript{IR}=0 or MSR\textsubscript{DR}=0, the interrupt is taken as if LPCR\textsubscript{AIL}=0.

**Use Process Table (UPRT)**

Controls whether Process Tables are used. For a radix-using partition, UPRT must be set to 1. For a paravirtualized HPT partition, UPRT is set to 1 when the operating system does not require the use of the legacy software-managed SLB.

- **0**: Process Table is not used. (Software-managed SLB in use, for paravirtualized HPT partition.)
- **1**: Process Table is used. (Segment Table in use, for paravirtualized HPT partition.)

**Programming Note**

The POWER9 processor operates as though LPCR\textsubscript{UPRT}=0 for partitions that use HPT translation, requiring operating systems to fully manage the SLB in software. Nonetheless, operating systems may need to maintain segment tables for use by accelerators.

**Enhanced Virtualization (EVIRT)**

Controls whether Enhanced Virtualization is enabled, as specified below.

- **0**: Enhanced Virtualization is disabled: attempts to execute hypervisor-privileged instructions or access hypervisor resources, or PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor resources, in privileged non-hypervisor state cause a Privileged Instruction type Program interrupt; attempts to access undefined SPR numbers (using mt\textsubscript{spr} or mfs\textsubscript{spr}) other than 0, 4, 5, and 6 in privileged state are treated as no-ops.

- **1**: Enhanced Virtualization is enabled: attempts to execute hypervisor-privileged instructions or access hypervisor resources, or PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor resources, in privileged non-hypervisor state cause a Hypervisor Emulation Assistance interrupt; attempts to access undefined SPR numbers (using mt\textsubscript{spr} or mfs\textsubscript{spr}) other than 0, 4, 5, and 6 in privileged state cause a Hypervisor Emulation Assistance interrupt.

**Programming Note**

Running with LPC\textsubscript{REVIRT}=1 facilitates support of nested hypervisors (hypervisors that run with MSR\textsubscript{HV} \textsubscript{PR}=0b00 and have their use of hypervisor resources virtualized by a higher level hypervisor); see the relevant Programming Note in Section 7.5.18, “Hypervisor Emulation Assistance Interrupt”. It also permits emulation of new SPRs on designs that do not support them in hardware.

All accesses to the reserved noop SPRs (808-811) are always treated as noops, independent of the value of EVIRT.

**Host Radix (HR)**

Indicates whether the hypervisor uses Radix Tree translation for the partition, as specified below.

- **0**: hypervisor uses HPT translation for this partition.
- **1**: hypervisor uses Radix Tree translation for this partition.

The hypervisor must program HR to match the Host Radix bit in the Partition Table Entry for the partition indicated by LPIDR. If the values do not match and the thread is not in hypervisor real addressing mode or ultravisor real addressing mode, the results are undefined.
**Programming Note**

HR is duplicated in the LPCR because there are times such as immediately after a partition swap when it is difficult for hardware to quickly access the PATE.

The translation mode for the hypervisor is the same as the translation mode of the partition the hypervisor is serving. This is necessary for consistent, well-defined behavior when a hypervisor concurrently serves partitions using both translation modes, and it creates a requirement that HR=1 in the PATE for LPID=0 when Radix Tree Translation partitions exist in the system, because of the effLPID construct. The architecture may refer to the translation mode of the hypervisor rather than the HR value for the partition when the relationship to the hypervisor matters.

44 Reserved

45 **Online** (ONL)

0 The PURR and SPURR do not increment.
1 The PURR and SPURR increment.

**Programming Note**

Typically, the hypervisor sets the ONL bit to 0 when the thread is not in a power-saving mode, is not performing useful work, and is available for use. The hypervisor may take the state of the ONL bit into account when making course-grain load balancing and power management decisions.

46 **Large Decrementer** (LD)

0 Large Decrementer mode is not enabled.
1 Large Decrementer mode is enabled.

See Section 8.4 for additional information.

47:51 **Power-saving mode Exit Cause Enable (Lower Section)** (PECE<sub>L</sub>)

47 **Privileged Doorbell Exit Enable**

0 When the **stop** instruction is executed with PSSCREC=1, Directed Privileged Doorbell exceptions are not enabled to cause exit from power-saving mode.
1 When the **stop** instruction is executed with PSSCREC=1, Directed Privileged Doorbell exceptions are enabled to cause exit from power-saving mode.

48 **Hypervisor Doorbell Exit Enable**

0 When the **stop** instruction is executed with PSSCREC=1, Directed Hypervisor Doorbell exceptions are not enabled to cause exit from power-saving mode.
1 When the **stop** instruction is executed with PSSCREC=1, Directed Hypervisor Doorbell exceptions are enabled to cause exit from power-saving mode.

49 **External Exit Enable**

0 When the **stop** instruction is executed with PSSCREC=1, External exceptions are not enabled to cause exit from power-saving mode.
1 When the **stop** instruction is executed with PSSCREC=1, External exceptions are enabled to cause exit from power-saving mode.

50 **Decrementer Exit Enable**

0 When the **stop** instruction is executed with PSSCREC=1, Decrementer exceptions are not enabled to cause exit from power-saving mode.
1 When the **stop** instruction is executed with PSSCREC=1, Decrementer exceptions are enabled to cause exit from power-saving mode. (Decrementer exceptions do not occur if the state of the Decrementer is not maintained and updated as if the thread was not in power-saving mode.)

51 **Other Exit Enable**

0 When the **stop** instruction is executed with PSSCREC=1, Machine Check, Hypervisor Maintenance, and certain implementation-specific exceptions are not enabled to cause exit from power-saving mode.
1 When the **stop** instruction is executed with PSSCREC=1, Machine Check, Hypervisor Maintenance, and certain implementation-specific exceptions are enabled to cause exit from power-saving mode.

If the state of the PECE field is lost during power-saving mode, implementations must provide the means to exit power-saving mode upon the occurrence of a System Reset exception and any of the exceptions that were enabled by the PECE field when the **stop** instruction was executed. In addition, they may also exit power-saving mode on exceptions that were disabled by the PECE field as well. See Section 7.5.1 and Section 7.5.2 for additional information about exit from power-saving mode.

52 **Mediated External Exception Request** (MER)

0 A Mediated External exception is not requested.
A Mediated External exception is requested.

The exception effects of this bit are said to be consistent with the contents of this bit if one of the following statements is true.
- LPCRMER = 1 and a Mediated External exception exists.
- LPCRMER = 0 and a Mediated External exception does not exist.

A context synchronizing instruction or event that is executed or occurs when LPCRMER = 0 ensures that the exception effects of LPCRMER are consistent with the contents of LPCRMER. Otherwise, when an instruction changes the contents of LPCRMER, the exception effects of LPCRMER become consistent with the new contents of LPCRMER reasonably soon after the change.

**Programming Note**

LPCRMER provides a means for the hypervisor to direct an external exception to a partition independent of the partition's MSR_EE setting. (When MSR_EE=0, it is inappropriate for the hypervisor to deliver the exception.) Using LPCRMER, the partition can be interrupted upon enabling external interrupts. Without using LPCRMER, the hypervisor must check the state of MSR_EE whenever it gets control, which will result in less timely delivery of the exception to the partition.

Guest Translation Shootdown Enable (GTSE)

Controls whether the operating system is permitted to use tlbie, slbieg, and slbiag directly, or must issue a system call to the hypervisor.
- 0 Guest is not permitted to use tlbie, slbieg, slbiag, tlbsync, and slbsync.
- 1 Guest is permitted to use tlbie, slbieg, slbiag, tlbsync, and slbsync.

**Programming Note**

An operating system that uses HPT translation must know whether VPM is active in order to invalidate the translation for a specific page using tlbie[]. See the related Programming Notes in the descriptions of tlbie and tlbiel.

Translation Control (TC)

0 The secondary Page Table search is enabled.
1 The secondary Page Table search is disabled.

**Reserved**

Hypervisor External Interrupt Control (HEIC)

0 Direct External interrupts can occur in hypervisor state.
1 Direct External interrupts cannot occur in hypervisor state.

**Programming Note**

By setting HEIC=1, the Hypervisor Interrupt Virtualization handler can prevent External interrupts from occurring during the Hypervisor Virtualization interrupt handler. See Section 7.5.7.1.

Logical Partitioning Environment Selector (LPES)

0 External interrupts set the HSRRs, set MSR_HV to 1, and leave MSR_RI unchanged.
1 External interrupts set the SRRs, set MSR_RI to 0, and leave MSR_HV unchanged.

**Programming Note**

LPES = 1 should be used by operating systems not running under a hypervisor, so that external interrupts are directed to the SRRs rather than to the HSRRs.

Hypervisor Virtualization Interrupt Conditionally Enable (HVICE)

0 Hypervisor Virtualization interrupts are disabled.
1 Hypervisor Virtualization interrupts are enabled if permitted by MSR_EE, MSR/fwlink, and MSR_HV; see Section 7.5.21.

Hypervisor Decrementer Interrupt Conditionally Enable (HDICE)

0 Hypervisor Decrementer interrupts are disabled.
1 Hypervisor Decrementer interrupts are enabled if permitted by MSR_EE, MSR_HV, and MSR_HV; see Section 7.5.12 on page 1275.
See Section 7.5 on page 1260 for a description of how the setting of LPES affects the processing of interrupts.

### 2.3 Hypervisor Real Mode Offset Register (HRMOR)

The layout of the Hypervisor Real Mode Offset Register (HRMOR) is shown in Figure 1 below.

#### Figure 1. Hypervisor Real Mode Offset Register

All other fields are reserved.

The supported HRMO values are the non-negative multiples of $2^r$, where $r$ is an implementation-dependent value and $12 \leq r \leq 26$.

The contents of the HRMOR affect how some storage accesses are performed as described in Section 6.7.3 on page 1180 and Section 6.7.5 on page 1184.

### 2.4 Logical Partition Identification Register (LPIDR)

The layout of the Logical Partition Identification Register (LPIDR) is shown in Figure 2 below.

#### Figure 2. Logical Partition Identification Register

The contents of the LPIDR identify the partition to which the thread is assigned, affecting some aspects of translation. The number of LPIDR bits supported is implementation-dependent.

### 2.5 Processor Compatibility Register (PCR)

The layout of the Processor Compatibility Register (PCR) is shown in Figure 3 below.

#### Figure 3. Processor Compatibility Register

High-order PCR bits are assigned to control the availability of facilities. Low-order PCR bits are assigned to control the availability of resources that are new in a specified version of the Architecture. The two types of bits can interact. For example, if a facility is created in one version and extended in the next, the high-order bit enables the portion of the facility that was defined in the version of the architecture enabled in the low-order bits.

Each defined bit in the PCR controls whether certain instructions, SPRs, and other related facilities are available in problem state. Except as specified elsewhere in this section, the PCR has no effect on facilities when the thread is not in problem state, or on privileged facilities when the thread is in problem state. Facilities that are made unavailable by the PCR are treated as follows when the thread is in problem state:

- Except for `mffsl`, `mffsce`, `mffsdn`, `mffsdrn`, `mffsm`, and `mffsni`, non-privileged instructions are treated as illegal instructions.
- `mffsl`, `mffsce`, `mffsdn`, `mffsdrn`, `mffsm`, and `mffsni` perform as if they are an `mffs` instruction.
- SPRs are treated as if they were not defined for the implementation.
- The "reserved SPRs" (see Section 1.3.3 of Book I) are treated as not defined for the implementation.
- Fields in instructions are treated as if they were 0s except for the `L` field in `paste`.
- Values of fields in instructions cause the instruction to be treated as an invalid form of the instruction.
- Unless the third item of this list applies, bits in system registers read back 0s for `mfspr` and `mtspr` operations have no effect on their values, except as described immediately below for bits 44:45 of the XER.

For bits 44:45 of the XER, two pairs of bits are provided, an "OV32-CA32" bit pair for `XEROV32` and `XERCA32` and a "reserved" bit pair for legacy XER bits 44:45 behavior.

Which bit pair is read by `mfxer` is controlled by the PCR. `mtxer` writes to both bit pairs, independent of the PCR. `mcrxr` reads the "OV32-CA32" bit pair.
Each bit in the “OV32-CA32” bit pair is implicitly set by instructions that implicitly set their respective XER_{OV} or XER_{CA}, independent of the PCR. The “reserved” bit pair for bits 44:45 of the XER are not altered by these instructions, independent of the PCR.

Programming Note

The “reserved” bit pair does not conform to the usual rules for reading (mfspr) reserved bits in registers (see Section 1.3.3 of Book I) because some early implementations used bits 44:45 of the XER for implementation-specific purposes. On these implementations, and on subsequent implementations that implemented versions of the architecture that precede V. 3.0, mfxer returned the contents of the bits, despite that the bits were defined as reserved.

A defined bit in the PCR may also control whether certain instructions, SPRs, and other related facilities are available in a privileged state (MSR{\text{PR}=0}). Affected facilities will be specifically annotated.

Programming Note

When a bit in a system register is made unavailable by the PCR, mtspr operations performed on the register in problem state have no effect on the value of the bit regardless of the privilege state in which the register may subsequently be read.

A PCR bit may also determine how an instruction field value is interpreted or may define other behavior as specified in the bit definitions below.

As described in more detail below, the value 1 in a defined bit in the PCR makes the affected resources unavailable and the value 0 makes them available.

The initial state of the PCR is all 0s. In this state, all instructions and facilities supported by the processor are available in all privilege states.

Programming Note

Facilities that can be disabled in problem state by the PCR that also have enable bits in either the MSR or [H]FSCR include the BHRB instructions, event-based branch instructions, TAR, DSCR at SPR 3, SIER, MMCR2, the event-based branch instructions, and certain Floating-Point, Vector, and VSX instructions. When any of these facilities are made unavailable in problem state by the PCR, the corresponding [Hypervisor] Facility Unavailable, Floating-Point Unavailable, Vector, or VSX unavailable interrupts do not occur when the facility is accessed in problem state. Note, however, that the PCR does not affect privileged accesses, and thus any Hypervisor Facility Unavailable, Floating-Point Unavailable, Vector unavailable, or VSX unavailable interrupts that are specified to occur as a result of privileged accesses occur regardless of the PCR value.

Programming Note

Hypervisors written for a given version of the architecture generally cannot support facilities that are defined in a subsequent version of the architecture if those facilities have new state that would need to be preserved across context switches. (In the absence of such state, support may or may not be possible.) Therefore, hypervisors will set to 1 all PCR bits that are reserved in the given version of the architecture.

The PCR has no effect on the setting of the MSR and [H]SRR1 by interrupts (and of the Count Register by the System Call Vectored interrupt), and by the rfscv.
Because the PCR has no effect on privileged instructions except as specified above, privileged instructions that are available on newer implementations but not available on older implementations will behave differently when the thread is in problem state. On older implementations, a Hypervisor Emulation Assistance interrupt will occur because the instruction is undefined; on newer implementations, a Privileged Instruction type Program interrupt will occur because the instruction is implemented.

In future versions of the architecture, in general the lowest-order reserved bit of the PCR will be used to control the availability of the instructions and related resources that are new in that version of the architecture; the name of the bit will correspond to the previous version of the architecture (i.e., the newest version in which the instructions and related resources were not available).

In these future versions of the architecture, there will be a requirement that if any bit of the low-order defined bits is set to 1 then all higher-order bits of the defined low-order bits must also be set to 1, and the architecture version with which the implementation appears to comply, in problem state, will be the version corresponding to the name of the lowest-order 1 bit in the set of defined low-order PCR bits, or the current architecture version if none of these bits are 1. Also, in general the highest-order reserved bits will be used to control the availability of sets of instructions and related resources having the requirement that their availability be independent of versions of the architecture.
The bit definitions for the PCR are shown below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:2</td>
<td>Reserved</td>
</tr>
<tr>
<td>3</td>
<td>Matrix-Multiply Accelerator (MMA)</td>
</tr>
<tr>
<td></td>
<td>This bit controls the availability of the instructions listed in Table 1 on page 1130 for all privilege states.</td>
</tr>
<tr>
<td></td>
<td>0  The instructions listed in Table 1 are available if the v3.0 bit is also set to 0.</td>
</tr>
<tr>
<td></td>
<td>1  The instructions listed in Table 1 are not available.</td>
</tr>
<tr>
<td>4:58</td>
<td>Reserved</td>
</tr>
<tr>
<td>59</td>
<td>Version 3.0 (v3.0)</td>
</tr>
<tr>
<td></td>
<td>When MSRPR=1 (i.e., problem state), this bit controls the availability of the following instructions, facilities, and behaviors that were newly available in problem state in the version of the architecture subsequent to Version 3.0.</td>
</tr>
<tr>
<td></td>
<td>- The instructions listed in Table 2 on page 1131</td>
</tr>
<tr>
<td></td>
<td>- The value 0 in bit 10 of the paste instruction. When this value is unavailable, attempt to execute a paste instruction with bit 10 set to 0 causes a Hypervisor Emulation Assistance interrupt in all privilege levels. (Although the L field is new, it has behavior more like a new value of a field because its pre-existing value was 1 instead of 0, and there is a need to treat the new value as causing an illegal instruction behavior instead of being ignored.)</td>
</tr>
<tr>
<td></td>
<td>- The ability to perform a data transfer from one storage location to another or to another system. When this function is unavailable, attempting to use it in any privilege state causes a DSI or HDSI on the paste instruction, setting [H]DSISR\textsubscript{60} as would occur under Version 3.0 of the architecture. See the latest revision of Version 3.0 of the architecture for details.</td>
</tr>
<tr>
<td></td>
<td>- SIER2, SIER3 and MMCR3</td>
</tr>
<tr>
<td></td>
<td>0  The instructions, behaviors, and facilities listed above are available.</td>
</tr>
<tr>
<td></td>
<td>Any word in storage fetched as an instruction having a primary opcode of 0b0000001 is treated as the prefix of a prefixed instruction and the next sequential word in storage is treated as the suffix of that prefixed instruction.</td>
</tr>
<tr>
<td></td>
<td>1  The instructions, behaviors, and facilities listed above are unavailable.</td>
</tr>
</tbody>
</table>

Any word in storage fetched as an instruction having a primary opcode of 0b0000001 is treated as an illegal word instruction.

Because this bit affects whether prefixed instructions are treated as illegal word instructions, it may affect how the Hypervisor Emulation Assist interrupt sets HSRR\textsubscript{134} and HEIR. If this bit is set to 1, then the MMA bit must also be set to 1.

When MSRPR=1 and MMCR0\textsubscript{PMCC}=0b00, this bit controls whether read permission on Group B Performance Monitor registers (i.e., SIER, SIAR, SDAR, and MMCR\textsubscript{1} at SPR numbers 768, 780, 781 and 782, respectively) specified in Version 3.0 of the architecture is further conditional on MMCR\textsubscript{0 PMCC\textsubscript{EXT}} bit or not.

- 0 mfspr availability on the mentioned registers is conditional on MMCR\textsubscript{0 PMCC\textsubscript{EXT}} bit. |
| 1  mfspr on the mentioned registers is available in problem state without further conditions. |

Version 2.07 (v2.07)

When MSR\textsubscript{PR}=1 (i.e., problem state), this bit controls the availability of the following instructions, facilities, and behaviors that were newly available in problem state in the version of the architecture subsequent to Version 2.07.

- The instructions listed in Table 3 on page 1134 |
| scv |
| 0  The instructions, behaviors, and facilities listed above are available. |
| mfxer reads the contents of the “OV32-CA32” bit pair for XER bits 44:45. |
| 1  The instructions, behaviors, and facilities listed above are unavailable. |
| mfxer reads the contents of the “reserved” bit pair for XER bits 44:45. |
| 0  mcrxrx is available. |
| mfxer reads the contents of the “OV32-CA32” bit pair for XER bits 44:45. |
| 1  mcrxrx is unavailable. |
**mfxer** reads the contents of the "reserved" bit pair for XER bits 44:45. If this bit is set to 1, then the v3.0 bit must also be set to 1.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfxvb16ger2</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update)</td>
</tr>
<tr>
<td>mfxvb16ger2nn</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) (Negative Multiply, Negative Accumulate)</td>
</tr>
<tr>
<td>mfxvb16ger2np</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>mfxvb16ger2p</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>mxfv16ger2</td>
<td>Prefixed Masked VSX Vector Half-Precision GER (Rank-2 Update)</td>
</tr>
<tr>
<td>mxfv16ger2nn</td>
<td>Prefixed Masked VSX Vector Half-Precision GER (Rank-2 Update) (Negative Multiply, Negative Accumulate)</td>
</tr>
<tr>
<td>mxfv16ger2np</td>
<td>Prefixed Masked VSX Vector Half-Precision GER (Rank-2 Update) (Negative Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>mxfv16ger2p</td>
<td>Prefixed Masked VSX Vector Half-Precision GER (Rank-2 Update) (Positive Multiply, Negative Accumulate)</td>
</tr>
<tr>
<td>mxfv16ger2pp</td>
<td>Prefixed Masked VSX Vector Half-Precision GER (Rank-2 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>mxfv32ger</td>
<td>Prefixed Masked VSX Vector Single-Precision GER (Rank-1 Update)</td>
</tr>
<tr>
<td>mxfv32gernn</td>
<td>Prefixed Masked VSX Vector Single-Precision GER (Rank-1 Update) (Negative Multiply, Negative Accumulate)</td>
</tr>
<tr>
<td>mxfv32gerpp</td>
<td>Prefixed Masked VSX Vector Single-Precision GER (Rank-1 Update) (Negative Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>mxfv32gerpp</td>
<td>Prefixed Masked VSX Vector Single-Precision GER (Rank-1 Update) (Positive Multiply, Negative Accumulate)</td>
</tr>
<tr>
<td>mxfv32gerp</td>
<td>Prefixed Masked VSX Vector Single-Precision GER (Rank-1 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>mxfv64ger</td>
<td>Prefixed Masked VSX Vector Double-Precision GER (Rank-1 Update)</td>
</tr>
<tr>
<td>mxfv64gerpp</td>
<td>Prefixed Masked VSX Vector Double-Precision GER (Rank-1 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
</tbody>
</table>

**Table 1: v3.1 instructions controlled by the MMA bit**

(Sheet 1 of 4)
Table 1: v3.1 instructions controlled by the MMA bit
(Sheet 4 of 4)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvf16ger2np</td>
<td>VSX Vector Half-Precision GER (Rank-2 Update) (Negative Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>xvf16ger2pn</td>
<td>VSX Vector Half-Precision GER (Rank-2 Update) (Positive Multiply, Negative Accumulate)</td>
</tr>
<tr>
<td>xvf16ger2pp</td>
<td>VSX Vector Half-Precision GER (Rank-2 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>xvf32ger/germnn</td>
<td>VSX Vector Single-Precision GER (Rank-1 Update) (Negative Multiply, Negative Accumulate)</td>
</tr>
<tr>
<td>xvf32ger/mpp</td>
<td>VSX Vector Single-Precision GER (Rank-1 Update) (Negative Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>xvf32ger/pn</td>
<td>VSX Vector Single-Precision GER (Rank-1 Update) (Positive Multiply, Negative Accumulate)</td>
</tr>
<tr>
<td>xvf32ger/pp</td>
<td>VSX Vector Single-Precision GER (Rank-1 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>xvf64ger</td>
<td>VSX Vector Double-Precision GER (Rank-1 Update)</td>
</tr>
<tr>
<td>xvf64ger/mn</td>
<td>VSX Vector Double-Precision GER (Rank-1 Update) (Negative Multiply, Negative Accumulate)</td>
</tr>
<tr>
<td>xvf64ger/mpp</td>
<td>VSX Vector Double-Precision GER (Rank-1 Update) (Negative Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>xvf64ger/pn</td>
<td>VSX Vector Double-Precision GER (Rank-1 Update) (Positive Multiply, Negative Accumulate)</td>
</tr>
<tr>
<td>xvf64ger/pp</td>
<td>VSX Vector Double-Precision GER (Rank-1 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>xvi16ger2</td>
<td>VSX Vector Signed Halfword GER (Rank-2 Update)</td>
</tr>
<tr>
<td>xvi16ger2pp</td>
<td>VSX Vector Signed Halfword GER (Rank-2 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>xvi16ger2s</td>
<td>VSX Vector Signed Halfword GER (Rank-2 Update) with Saturate</td>
</tr>
<tr>
<td>xvi16ger2spp</td>
<td>VSX Vector Signed Halfword GER (Rank-2 Update) with Saturate (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>xvi4ger8</td>
<td>VSX Vector Signed Nibble GER (Rank-8 Update)</td>
</tr>
<tr>
<td>xvi4ger8pp</td>
<td>VSX Vector Signed Nibble GER (Rank-8 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>xvi8ger4</td>
<td>VSX Vector Signed/Unsigned Byte GER (Rank-4 Update)</td>
</tr>
<tr>
<td>xvi8ger4pp</td>
<td>VSX Vector Signed/Unsigned Byte GER (Rank-4 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>xvi8ger4ppm</td>
<td>VSX Vector Signed/Unsigned Byte GER (Rank-4 Update) (Positive Multiply, Positive Accumulate)</td>
</tr>
<tr>
<td>xxmfa</td>
<td>VSX Move From ACC</td>
</tr>
<tr>
<td>xxmtacc</td>
<td>VSX Move To ACC</td>
</tr>
<tr>
<td>xxsetaccz</td>
<td>VSX Set ACC to Zero</td>
</tr>
</tbody>
</table>

Table 2: Instructions controlled by the v3.0 bit
(Sheet 1 of 6)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brd</td>
<td>Byte-Reverse Doubleword</td>
</tr>
<tr>
<td>brh</td>
<td>Byte-Reverse Halfword</td>
</tr>
<tr>
<td>brw</td>
<td>Byte-Reverse Word</td>
</tr>
<tr>
<td>cfuged</td>
<td>Centrifuge Doubleword</td>
</tr>
<tr>
<td>cntldm</td>
<td>Count Leading Zeros Doubleword under bit Mask</td>
</tr>
<tr>
<td>cnttldm</td>
<td>Count Trailing Zeros Doubleword under bit Mask</td>
</tr>
<tr>
<td>dcflixq</td>
<td>DFP Convert From Fixed Quadword Quad</td>
</tr>
<tr>
<td>dcflixq</td>
<td>DFP Convert To Fixed Quadword Quad</td>
</tr>
<tr>
<td>lxvkq</td>
<td>Load VSX Vector Special Value Quad</td>
</tr>
<tr>
<td>lxvp</td>
<td>Load VSX Vector Paired</td>
</tr>
<tr>
<td>lxvpx</td>
<td>Load VSX Vector Paired Indexed</td>
</tr>
<tr>
<td>lxvrbx</td>
<td>Load VSX Vector Rightmost Byte Indexed</td>
</tr>
<tr>
<td>lxvrdx</td>
<td>Load VSX Vector Rightmost Doubleword Indexed</td>
</tr>
<tr>
<td>lxvrxh</td>
<td>Load VSX Vector Rightmost Halfword Indexed</td>
</tr>
<tr>
<td>lxvnx</td>
<td>Load VSX Vector Rightmost Word Indexed</td>
</tr>
<tr>
<td>mtvsrbm</td>
<td>Move to VSR Byte Mask</td>
</tr>
<tr>
<td>mtvsrbm</td>
<td>Move to VSR Byte Mask Immediate</td>
</tr>
<tr>
<td>mtvstdm</td>
<td>Move to VSR Doubleword Mask</td>
</tr>
<tr>
<td>mtvshrhm</td>
<td>Move to VSR Halfword Mask</td>
</tr>
<tr>
<td>mtvshrqm</td>
<td>Move to VSR Quadword Mask</td>
</tr>
<tr>
<td>mtvsvrwm</td>
<td>Move to VSR Word Mask</td>
</tr>
<tr>
<td>paddi</td>
<td>Prefixed Add Immediate</td>
</tr>
<tr>
<td>pdepd</td>
<td>Parallel Bits Deposit Doubleword</td>
</tr>
<tr>
<td>pextd</td>
<td>Parallel Bits Extract Doubleword</td>
</tr>
<tr>
<td>plbz</td>
<td>Prefixed Load Byte &amp; Zero</td>
</tr>
<tr>
<td>plfd</td>
<td>Prefixed Load Floating Double</td>
</tr>
<tr>
<td>pilsf</td>
<td>Prefixed Load Floating Single</td>
</tr>
</tbody>
</table>
Table 2: Instructions controlled by the v3.0 bit
(Sheet 2 of 6)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plha</td>
<td>Prefixed Load Halfword Algebraic</td>
</tr>
<tr>
<td>plhz</td>
<td>Prefixed Load Halfword &amp; Zero</td>
</tr>
<tr>
<td>plq</td>
<td>Prefixed Load Quadword</td>
</tr>
<tr>
<td>plwa</td>
<td>Prefixed Load Word Algebraic</td>
</tr>
<tr>
<td>plwz</td>
<td>Prefixed Load Word &amp; Zero</td>
</tr>
<tr>
<td>plxsd</td>
<td>Prefixed Load VSX Scalar Doubleword</td>
</tr>
<tr>
<td>plxspp</td>
<td>Prefixed Load VSX Scalar Single</td>
</tr>
<tr>
<td>plxv</td>
<td>Prefixed Load VSX Vector</td>
</tr>
<tr>
<td>plxvp</td>
<td>Prefixed Load VSX Vector Paired</td>
</tr>
<tr>
<td>pnop</td>
<td>Prefixed No-Operation</td>
</tr>
<tr>
<td>pstd</td>
<td>Prefixed Store Doubleword</td>
</tr>
<tr>
<td>pstfd</td>
<td>Prefixed Store Byte</td>
</tr>
<tr>
<td>pstfs</td>
<td>Prefixed Store Single</td>
</tr>
<tr>
<td>psth</td>
<td>Prefixed Store Halfword</td>
</tr>
<tr>
<td>pstq</td>
<td>Prefixed Store Quadword</td>
</tr>
<tr>
<td>pstw</td>
<td>Prefixed Store Word</td>
</tr>
<tr>
<td>ptxsd</td>
<td>Prefixed Store VSX Scalar Doubleword</td>
</tr>
<tr>
<td>ptxspp</td>
<td>Prefixed Store VSX Scalar Single-Precision</td>
</tr>
<tr>
<td>ptxv</td>
<td>Prefixed Store VSX Vector</td>
</tr>
<tr>
<td>ptxvp</td>
<td>Prefixed Store VSX Vector Paired</td>
</tr>
<tr>
<td>setbc</td>
<td>Set Boolean Condition</td>
</tr>
<tr>
<td>setbcr</td>
<td>Set Boolean Condition Reverse</td>
</tr>
<tr>
<td>setbnc</td>
<td>Set Negative Boolean Condition</td>
</tr>
<tr>
<td>setbncr</td>
<td>Set Negative Boolean Condition Reverse</td>
</tr>
<tr>
<td>stxvp</td>
<td>Store VSX Vector Paired</td>
</tr>
<tr>
<td>stxvpx</td>
<td>Store VSX Vector Paired Indexed</td>
</tr>
<tr>
<td>stxvrbx</td>
<td>Store VSX Vector Rightmost Byte Indexed</td>
</tr>
<tr>
<td>stxvrdx</td>
<td>Store VSX Vector Rightmost Doubleword Indexed</td>
</tr>
<tr>
<td>stxvrhx</td>
<td>Store VSX Vector Rightmost Halfword Indexed</td>
</tr>
<tr>
<td>stxvrwx</td>
<td>Store VSX Vector Rightmost Word Indexed</td>
</tr>
<tr>
<td>vcfuged</td>
<td>Vector Centrifuge Doubleword</td>
</tr>
<tr>
<td>vclrb</td>
<td>Vector Clear Leftmost Bytes</td>
</tr>
<tr>
<td>vclrb</td>
<td>Vector Clear Rightmost Bytes</td>
</tr>
<tr>
<td>vclzdm</td>
<td>Vector Count Leading Zeros Doubleword under bit Mask</td>
</tr>
<tr>
<td>vcmpequq[.]</td>
<td>Vector Compare Equal Quadword</td>
</tr>
<tr>
<td>vcmptgsa[.]</td>
<td>Vector Compare Greater Than Signed Quadword</td>
</tr>
<tr>
<td>vcmptguq[.]</td>
<td>Vector Compare Greater Than Unsigned Quadword</td>
</tr>
<tr>
<td>vcmpsq</td>
<td>Vector Compare Signed Quadword</td>
</tr>
<tr>
<td>vcmpsq</td>
<td>Vector Compare Unsigned Quadword</td>
</tr>
<tr>
<td>vcntmbb</td>
<td>Vector Count Mask Bits Byte</td>
</tr>
<tr>
<td>vcntmbd</td>
<td>Vector Count Mask Bits Doubleword</td>
</tr>
<tr>
<td>vcntmbh</td>
<td>Vector Count Mask Bits Halfword</td>
</tr>
<tr>
<td>vcntmbw</td>
<td>Vector Count Mask Bits Word</td>
</tr>
<tr>
<td>vctzdm</td>
<td>Vector Count Trailing Zeros Doubleword under bit Mask</td>
</tr>
<tr>
<td>vdivesd</td>
<td>Vector Divide Extended Signed Doubleword</td>
</tr>
<tr>
<td>vdivesq</td>
<td>Vector Divide Extended Signed Quadword</td>
</tr>
<tr>
<td>vdivesw</td>
<td>Vector Divide Extended Signed Word</td>
</tr>
<tr>
<td>vdivued</td>
<td>Vector Divide Extended Unsigned Doubleword</td>
</tr>
<tr>
<td>vdivueq</td>
<td>Vector Divide Extended Unsigned Quadword</td>
</tr>
<tr>
<td>vdivuw</td>
<td>Vector Divide Extended Unsigned Word</td>
</tr>
<tr>
<td>vdivsd</td>
<td>Vector Divide Signed Doubleword</td>
</tr>
<tr>
<td>vdivsq</td>
<td>Vector Divide Signed Quadword</td>
</tr>
<tr>
<td>vdivsw</td>
<td>Vector Divide Signed Word</td>
</tr>
<tr>
<td>vdivud</td>
<td>Vector Divide Unsigned Doubleword</td>
</tr>
<tr>
<td>vdivuq</td>
<td>Vector Divide Unsigned Quadword</td>
</tr>
<tr>
<td>vdivuw</td>
<td>Vector Divide Unsigned Word</td>
</tr>
<tr>
<td>vexpandbm</td>
<td>Vector Expand Byte Mask</td>
</tr>
<tr>
<td>vexpanddm</td>
<td>Vector Expand Doubleword Mask</td>
</tr>
<tr>
<td>vexpandhm</td>
<td>Vector Expand Halfword Mask</td>
</tr>
<tr>
<td>vexpandqm</td>
<td>Vector Expand Quadword Mask</td>
</tr>
<tr>
<td>vexpandwm</td>
<td>Vector Expand Word Mask</td>
</tr>
<tr>
<td>vextddvix</td>
<td>Vector Extract Double Doubleword to VSR Left-Indexed</td>
</tr>
<tr>
<td>vextddvrx</td>
<td>Vector Extract Double Doubleword to VSR Right-Indexed</td>
</tr>
<tr>
<td>vextdubvix</td>
<td>Vector Extract Double Unsigned Byte to Vector Register Left-Indexed</td>
</tr>
<tr>
<td>vextduvrx</td>
<td>Vector Extract Double Unsigned Byte to Vector Register Right-Indexed</td>
</tr>
<tr>
<td>vextduvix</td>
<td>Vector Extract Double Unsigned Halfword to Vector Register Left-Indexed</td>
</tr>
<tr>
<td>vextduvrx</td>
<td>Vector Extract Double Unsigned Halfword to Vector Register Right-Indexed</td>
</tr>
<tr>
<td>vextduwvix</td>
<td>Vector Extract Double Unsigned Word to Vector Register Left-Indexed</td>
</tr>
<tr>
<td>vextduwvrx</td>
<td>Vector Extract Double Unsigned Word to Vector Register Right-Indexed</td>
</tr>
<tr>
<td>vextrdm</td>
<td>Vector Extract Doubleword Mask</td>
</tr>
<tr>
<td>vextrdm</td>
<td>Vector Extract Doubleword Mask</td>
</tr>
<tr>
<td>vextrdm</td>
<td>Vector Extract Doubleword Mask</td>
</tr>
<tr>
<td>vextrdm</td>
<td>Vector Extract Doubleword Mask</td>
</tr>
<tr>
<td>vextrdm</td>
<td>Vector Extract Doubleword Mask</td>
</tr>
</tbody>
</table>

Table 2: Instructions controlled by the v3.0 bit
(Sheet 3 of 6)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgnrb</td>
<td>Vector Gather every Nth Bit</td>
</tr>
<tr>
<td>vinsblx</td>
<td>Vector Insert Byte from GPR Left-Indexed</td>
</tr>
<tr>
<td>vinsbrx</td>
<td>Vector Insert Byte from GPR Right-Indexed</td>
</tr>
<tr>
<td>vinsbvblx</td>
<td>Vector Insert Byte from VSR Left-Indexed</td>
</tr>
<tr>
<td>vinsbvrex</td>
<td>Vector Insert Byte from VSR Right-Indexed</td>
</tr>
<tr>
<td>vinds</td>
<td>Vector Insert Doubleword from GPR</td>
</tr>
<tr>
<td>vindslix</td>
<td>Vector Insert Doubleword from GPR Left-Indexed</td>
</tr>
<tr>
<td>vindsrxx</td>
<td>Vector Insert Doubleword from GPR Right-Indexed</td>
</tr>
<tr>
<td>vinshlx</td>
<td>Vector Insert Halfword from GPR Left-Indexed</td>
</tr>
<tr>
<td>vinshrx</td>
<td>Vector Insert Halfword from GPR Right-Indexed</td>
</tr>
<tr>
<td>vinshvblx</td>
<td>Vector Insert Halfword from VSR Left-Indexed</td>
</tr>
<tr>
<td>vinshvrex</td>
<td>Vector Insert Halfword from VSR Right-Indexed</td>
</tr>
<tr>
<td>vinsw</td>
<td>Vector Insert Word from GPR</td>
</tr>
<tr>
<td>vinswlix</td>
<td>Vector Insert Word from GPR Left-Indexed</td>
</tr>
<tr>
<td>vinswrxx</td>
<td>Vector Insert Word from GPR Right-Indexed</td>
</tr>
<tr>
<td>vinswwblx</td>
<td>Vector Insert Word from VSR Left-Indexed</td>
</tr>
<tr>
<td>vinswwrex</td>
<td>Vector Insert Word from VSR Right-Indexed</td>
</tr>
<tr>
<td>vmodsd</td>
<td>Vector Modulo Signed Doubleword</td>
</tr>
<tr>
<td>vmodsq</td>
<td>Vector Modulo Signed Quadword</td>
</tr>
<tr>
<td>vmodsw</td>
<td>Vector Modulo Signed Word</td>
</tr>
<tr>
<td>vmodud</td>
<td>Vector Modulo Unsigned Doubleword</td>
</tr>
<tr>
<td>vmoduq</td>
<td>Vector Modulo Unsigned Quadword</td>
</tr>
<tr>
<td>vmoduw</td>
<td>Vector Modulo Unsigned Word</td>
</tr>
<tr>
<td>vmsumcud</td>
<td>Vector Multiply-Sum &amp; write Carry-out Uns signed Doubleword</td>
</tr>
<tr>
<td>vmulesd</td>
<td>Vector Multiply Even Signed Doubleword</td>
</tr>
<tr>
<td>vmuleud</td>
<td>Vector Multiply Even Unsigned Doubleword</td>
</tr>
<tr>
<td>vmulhsd</td>
<td>Vector Multiply High Signed Doubleword</td>
</tr>
<tr>
<td>vmulhsw</td>
<td>Vector Multiply High Signed Word</td>
</tr>
<tr>
<td>vmulhusd</td>
<td>Vector Multiply High Signed Unsigned Doubleword</td>
</tr>
<tr>
<td>vmulhwd</td>
<td>Vector Multiply High Unsigned Doubleword</td>
</tr>
<tr>
<td>vmulhwsd</td>
<td>Vector Multiply High Unsigned Word</td>
</tr>
<tr>
<td>vmullds</td>
<td>Vector Multiply Low Doubleword</td>
</tr>
<tr>
<td>vmulosd</td>
<td>Vector Multiply Odd Signed Doubleword</td>
</tr>
<tr>
<td>vmuloud</td>
<td>Vector Multiply Odd Unsigned Doubleword</td>
</tr>
<tr>
<td>vpdepd</td>
<td>Vector Parallel Bits Deposit Doubleword</td>
</tr>
<tr>
<td>vpxtd</td>
<td>Vector Parallel Bits Extract Doubleword</td>
</tr>
<tr>
<td>vrlq</td>
<td>Vector Rotate Left Quadword</td>
</tr>
<tr>
<td>vrlqmi</td>
<td>Vector Rotate Left Quadword then Insert</td>
</tr>
<tr>
<td>vrlqnm</td>
<td>Vector Rotate Left Quadword then Mask And with Mask</td>
</tr>
<tr>
<td>vslbl</td>
<td>Vector Shift Left Byte by Bit Immediate</td>
</tr>
<tr>
<td>vsrc</td>
<td>Vector Shift Right Quadword</td>
</tr>
<tr>
<td>vsrcbl</td>
<td>Vector Shift Right Byte-justified</td>
</tr>
<tr>
<td>vsrcbr</td>
<td>Vector Shift Right Halword-justified</td>
</tr>
<tr>
<td>vsrcfr</td>
<td>Vector Shift Right Double by Bit Immediate</td>
</tr>
<tr>
<td>vsrcg</td>
<td>Vector Shift Right Quadword</td>
</tr>
<tr>
<td>vstribi</td>
<td>Vector String Isolate Byte Left-justified</td>
</tr>
<tr>
<td>vstribr</td>
<td>Vector String Isolate Byte Right-justified</td>
</tr>
<tr>
<td>vstrilh</td>
<td>Vector String Isolate Halfword Left-justified</td>
</tr>
<tr>
<td>vstrirh</td>
<td>Vector String Isolate Halfword Right-justified</td>
</tr>
<tr>
<td>xscmpeqpp</td>
<td>VSX Scalar Compare Equal Quad-Precision</td>
</tr>
<tr>
<td>xscmpgeqpp</td>
<td>VSX Scalar Compare Greater Than or Equal Quad-Precision</td>
</tr>
<tr>
<td>xscmpgtqp</td>
<td>VSX Scalar Compare Greater Than Quad-Precision</td>
</tr>
<tr>
<td>xscvqpsqz</td>
<td>VSX Vector Convert Quad-Precision to Signed Quadword</td>
</tr>
<tr>
<td>xscvqpuqz</td>
<td>VSX Vector Convert Quad-Precision to Unsigned Quadword</td>
</tr>
<tr>
<td>xscvsqpp</td>
<td>VSX Vector Convert Signed Quadword to Quad-Precision</td>
</tr>
<tr>
<td>xscvuqpp</td>
<td>VSX Vector Convert Unsigned Quadword to Quad-Precision</td>
</tr>
<tr>
<td>xsmaxcqp</td>
<td>VSX Scalar Maximum Type-C Quad-Precision</td>
</tr>
<tr>
<td>xsmincqp</td>
<td>VSX Scalar Minimum Type-C Quad-Precision</td>
</tr>
<tr>
<td>xvcvbf16sp</td>
<td>VSX Vector Convert float16 to Single-Precision format</td>
</tr>
<tr>
<td>xvcvspbf16</td>
<td>VSX Vector Convert with round Single-Precision to float16 format</td>
</tr>
<tr>
<td>xvtlsbb</td>
<td>VSX Vector Test Least-Significant Bit by Byte Operation</td>
</tr>
<tr>
<td>xxblendvb</td>
<td>VSX Vector Blend Variable Byte</td>
</tr>
<tr>
<td>xxblendvd</td>
<td>VSX Vector Blend Variable Doubleword</td>
</tr>
<tr>
<td>xxblendvh</td>
<td>VSX Vector Blend Variable Halffgeow</td>
</tr>
<tr>
<td>xxblendvw</td>
<td>VSX Vector Blend Variable Word</td>
</tr>
<tr>
<td>xxeval</td>
<td>VSX Vector Evaluate</td>
</tr>
<tr>
<td>xxgenpcvb</td>
<td>VSX Vector Generate PCV from Byte Mask</td>
</tr>
<tr>
<td>xxgenpcvdm</td>
<td>VSX Vector Generate PCV from Doubleword Mask</td>
</tr>
</tbody>
</table>

Table 2: Instructions controlled by the v3.0 bit
(Sheet 4 of 6)
Table 2: Instructions controlled by the v3.0 bit
(Sheet 6 of 6)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addpcis</td>
<td>Add PC Immediate Shifted Prefix</td>
</tr>
<tr>
<td>bcdcfn</td>
<td>Decimal Convert From National</td>
</tr>
<tr>
<td>bcdcfsq</td>
<td>Decimal Convert From Signed Quadword</td>
</tr>
<tr>
<td>bcdcfz</td>
<td>Decimal Convert From Zoned</td>
</tr>
<tr>
<td>bcdcpgsn</td>
<td>Decimal CopySign</td>
</tr>
<tr>
<td>bcdctn</td>
<td>Decimal Convert To National</td>
</tr>
<tr>
<td>bcdctsq</td>
<td>Decimal Convert To Signed Quadword</td>
</tr>
<tr>
<td>bcdctz</td>
<td>Decimal Convert To Zoned</td>
</tr>
<tr>
<td>bcds</td>
<td>Decimal Shift</td>
</tr>
<tr>
<td>bcddtsgn</td>
<td>Decimal Set Sign</td>
</tr>
<tr>
<td>bcddr</td>
<td>Decimal Shift and Round</td>
</tr>
<tr>
<td>bcddtrunc</td>
<td>Decimal Truncate</td>
</tr>
<tr>
<td>bcddus</td>
<td>Decimal Unsigned Shift</td>
</tr>
<tr>
<td>bcddtrunc</td>
<td>Decimal Unsigned Truncate</td>
</tr>
<tr>
<td>cmpceb</td>
<td>Compare Equal Byte</td>
</tr>
<tr>
<td>cmprb</td>
<td>Compare Ranged Byte</td>
</tr>
<tr>
<td>cnttzd[.]</td>
<td>Count Trailing Zeros Doubleword</td>
</tr>
<tr>
<td>cnttzw[.]</td>
<td>Count Trailing Zeros Word</td>
</tr>
<tr>
<td>copy</td>
<td>Copy</td>
</tr>
<tr>
<td>cpabort</td>
<td>Copy-Paste Abort</td>
</tr>
<tr>
<td>darr</td>
<td>Deliver a Random Number</td>
</tr>
<tr>
<td>dtstsfii</td>
<td>DFP Test Significance Immediate</td>
</tr>
<tr>
<td>dtstsfiqi</td>
<td>DFP Test Significance Immediate Quad</td>
</tr>
<tr>
<td>extswsli[.]</td>
<td>Extend Sign Word and Shift Left Immediate</td>
</tr>
<tr>
<td>ldad</td>
<td>Load Doubleword Atomic</td>
</tr>
<tr>
<td>ldat</td>
<td>Load Doubleword Atomic</td>
</tr>
<tr>
<td>lwat</td>
<td>Load Word Atomic</td>
</tr>
<tr>
<td>lxd</td>
<td>Load VSX Scalar Doubleword</td>
</tr>
<tr>
<td>lxsbicx</td>
<td>Load VSX Scalar as Integer Byte &amp; Zero Indexed</td>
</tr>
<tr>
<td>lxsihcx</td>
<td>Load VSX Scalar as Integer Halfword &amp; Zero Indexed</td>
</tr>
<tr>
<td>lxspsp</td>
<td>Load VSX Scalar Single</td>
</tr>
<tr>
<td>lxv</td>
<td>Load VSX Vector</td>
</tr>
</tbody>
</table>

Table 3: Instructions controlled by the v2.07 bit
(Sheet 1 of 6)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxvb16x</td>
<td>Load VSX Vector Byte*16 Indexed</td>
</tr>
<tr>
<td>lxvh8x</td>
<td>Load VSX Vector Halfword*8 Indexed</td>
</tr>
<tr>
<td>lxvl</td>
<td>Load VSX Vector with Length</td>
</tr>
<tr>
<td>lxvl</td>
<td>Load VSX Vector Left-justified with Length</td>
</tr>
<tr>
<td>lxwvssx</td>
<td>Load VSX Vector Word &amp; Splat Indexed</td>
</tr>
<tr>
<td>lxwx</td>
<td>Load VSX Vector Indexed</td>
</tr>
<tr>
<td>maddhd</td>
<td>Multiply-Add High Doubleword</td>
</tr>
<tr>
<td>maddhd</td>
<td>Multiply-Add High Doubleword Unsigned</td>
</tr>
<tr>
<td>maddld</td>
<td>Multiply-Add Low Doubleword</td>
</tr>
<tr>
<td>mcrrx</td>
<td>Move XER to CR Extended</td>
</tr>
<tr>
<td>mffscdmi</td>
<td>Move From FPSCR Control &amp; set DRN Immediate</td>
</tr>
<tr>
<td>mffsce</td>
<td>Move From FPSCR &amp; Clear Enables</td>
</tr>
<tr>
<td>mffscrmi</td>
<td>Move From FPSCR Control &amp; set RN Immediate</td>
</tr>
<tr>
<td>mffscr</td>
<td>Move From FPSCR Control</td>
</tr>
<tr>
<td>mffscl</td>
<td>Move From FPSCR Control &amp; set DRN</td>
</tr>
<tr>
<td>mffscl</td>
<td>Move From FPSCR Control</td>
</tr>
<tr>
<td>mffscl</td>
<td>Move From FPSCR Control &amp; set RN</td>
</tr>
<tr>
<td>mffscl</td>
<td>Move From FPSCR Control &amp; set DRN Immediate</td>
</tr>
<tr>
<td>mffvsrd</td>
<td>Move From VSR Lower Doubleword</td>
</tr>
<tr>
<td>modsd</td>
<td>Modulo Signed Doubleword</td>
</tr>
<tr>
<td>modsw</td>
<td>Modulo Signed Word</td>
</tr>
<tr>
<td>modud</td>
<td>Modulo Unsigned Doubleword</td>
</tr>
<tr>
<td>moduw</td>
<td>Modulo Unsigned Word</td>
</tr>
<tr>
<td>mtvsrdrd</td>
<td>Move To VSR Double Doubleword</td>
</tr>
<tr>
<td>mtvsrws</td>
<td>Move To VSR Word &amp; Splat</td>
</tr>
<tr>
<td>paste</td>
<td>Paste</td>
</tr>
<tr>
<td>setb</td>
<td>Set Boolean</td>
</tr>
<tr>
<td>stdat</td>
<td>Store Doubleword Atomic</td>
</tr>
<tr>
<td>stwat</td>
<td>Store Word Atomic</td>
</tr>
<tr>
<td>stxsd</td>
<td>Store VSX Scalar Doubleword</td>
</tr>
<tr>
<td>stxisb</td>
<td>Store VSX Scalar as Integer Byte Indexed</td>
</tr>
<tr>
<td>stxisihx</td>
<td>Store VSX Scalar as Integer Halfword Indexed</td>
</tr>
<tr>
<td>stxisp</td>
<td>Store VSX Scalar Single</td>
</tr>
<tr>
<td>stxv</td>
<td>Store VSX Vector</td>
</tr>
<tr>
<td>stxvb16x</td>
<td>Store VSX Vector Byte*16 Indexed</td>
</tr>
<tr>
<td>stxvhsx</td>
<td>Store VSX Vector Halfword*8 Indexed</td>
</tr>
<tr>
<td>stxvl</td>
<td>Store VSX Vector with Length</td>
</tr>
<tr>
<td>stxvll</td>
<td>Store VSX Vector Left-justified with Length</td>
</tr>
<tr>
<td>vabsdub</td>
<td>Vector Absolute Difference Unsigned Byte</td>
</tr>
<tr>
<td>vabsduh</td>
<td>Vector Absolute Difference Unsigned Halfword</td>
</tr>
<tr>
<td>vabsduw</td>
<td>Vector Absolute Difference Unsigned Word</td>
</tr>
</tbody>
</table>

Table 3: Instructions controlled by the v2.07 bit
(Sheet 2 of 6)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbpermdb</td>
<td>Vector Bit Permute Doubleword</td>
</tr>
<tr>
<td>vclzlbsbb</td>
<td>Vector Count Leading Zero Least-Significant Bits Byte</td>
</tr>
<tr>
<td>vcmpneb[b]</td>
<td>Vector Compare Not Equal Byte</td>
</tr>
<tr>
<td>vcmpneh[b]</td>
<td>Vector Compare Not Equal Halfword</td>
</tr>
<tr>
<td>vcmpnew[w]</td>
<td>Vector Compare Not Equal Word</td>
</tr>
<tr>
<td>vcmpnewz[b]</td>
<td>Vector Compare Not Equal or Zero Byte</td>
</tr>
<tr>
<td>vcmpnewzh[w]</td>
<td>Vector Compare Not Equal or Zero Halfword</td>
</tr>
<tr>
<td>vcptrb</td>
<td>Vector Count Trailing Zeros Byte</td>
</tr>
<tr>
<td>vcptrd</td>
<td>Vector Count Trailing Zeros Doubleword</td>
</tr>
<tr>
<td>vcptrzh</td>
<td>Vector Count Trailing Zeros Halfword</td>
</tr>
<tr>
<td>vctzlbsbb</td>
<td>Vector Count Trailing Zero Least-Significant Bits Byte</td>
</tr>
<tr>
<td>vctzw</td>
<td>Vector Count Trailing Zeros Word</td>
</tr>
<tr>
<td>vextractd</td>
<td>Vector Extract Doubleword</td>
</tr>
<tr>
<td>vextractub</td>
<td>Vector Extract Unsigned Byte</td>
</tr>
<tr>
<td>vextractuh</td>
<td>Vector Extract Unsigned Halfword</td>
</tr>
<tr>
<td>vextsbd2d</td>
<td>Vector Extend Sign Byte To Doubleword</td>
</tr>
<tr>
<td>vextsbd2w</td>
<td>Vector Extend Sign Byte To Word</td>
</tr>
<tr>
<td>vextsdh2d</td>
<td>Vector Extend Sign Halfword To Doubleword</td>
</tr>
<tr>
<td>vextsdh2w</td>
<td>Vector Extend Sign Halfword To Word</td>
</tr>
<tr>
<td>vextsw2d</td>
<td>Vector Extend Sign Word To Doubleword</td>
</tr>
<tr>
<td>vextublx</td>
<td>Vector Extract Unsigned Byte Left-Indexed</td>
</tr>
<tr>
<td>vextubrx</td>
<td>Vector Extract Unsigned Byte Right-Indexed</td>
</tr>
<tr>
<td>vextuhlx</td>
<td>Vector Extract Unsigned Halfword Left-Indexed</td>
</tr>
<tr>
<td>vextuhrx</td>
<td>Vector Extract Unsigned Halfword Right-Indexed</td>
</tr>
<tr>
<td>vextuwlx</td>
<td>Vector Extract Unsigned Word Left-Indexed</td>
</tr>
<tr>
<td>vextuwrx</td>
<td>Vector Extract Unsigned Word Right-Indexed</td>
</tr>
<tr>
<td>vinsetb</td>
<td>Vector Insert Byte</td>
</tr>
<tr>
<td>vinserd</td>
<td>Vector Insert Doubleword</td>
</tr>
<tr>
<td>vinserrh</td>
<td>Vector Insert Halfword</td>
</tr>
<tr>
<td>vinsertw</td>
<td>Vector Insert Word</td>
</tr>
<tr>
<td>vmul10cuq</td>
<td>Vector Multiply-by-10 &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>vmul10cuq</td>
<td>Vector Multiply-by-10 Extended &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>vmul10eq</td>
<td>Vector Multiply-by-10 Extended Unsigned Quadword</td>
</tr>
<tr>
<td>vmul10uq</td>
<td>Vector Multiply-by-10 Unsigned Quadword</td>
</tr>
<tr>
<td>vnegd</td>
<td>Vector Negate Doubleword</td>
</tr>
<tr>
<td>vnegw</td>
<td>Vector Negate Word</td>
</tr>
<tr>
<td>vpermr</td>
<td>Vector Permute Right-indexed</td>
</tr>
<tr>
<td>vptrbzd</td>
<td>Vector Parity Byte Doubleword</td>
</tr>
<tr>
<td>vptrbqz</td>
<td>Vector Parity Byte Quadword</td>
</tr>
<tr>
<td>vptrbwz</td>
<td>Vector Parity Byte Word</td>
</tr>
<tr>
<td>vrdmiz</td>
<td>Vector Rotate Left Doubleword then Mask Insert</td>
</tr>
<tr>
<td>vrdmn</td>
<td>Vector Rotate Left Doubleword then AND with Mask</td>
</tr>
<tr>
<td>vrlwmiz</td>
<td>Vector Rotate Left Word then Mask Insert</td>
</tr>
<tr>
<td>vrlwnm</td>
<td>Vector Rotate Left Word then AND with Mask</td>
</tr>
<tr>
<td>vslv</td>
<td>Vector Shift Left Variable</td>
</tr>
<tr>
<td>vsrv</td>
<td>Vector Shift Right Variable</td>
</tr>
<tr>
<td>wait</td>
<td>Wait</td>
</tr>
<tr>
<td>xsabsqp</td>
<td>VSX Scalar Quad-Precision Absolute</td>
</tr>
<tr>
<td>xsaddqp[a]</td>
<td>VSX Scalar Quad-Precision Add [&amp; round to Odd]</td>
</tr>
<tr>
<td>xscmpexpdp</td>
<td>VSX Scalar Double-Precision Compare Exponents</td>
</tr>
<tr>
<td>xscmpexpqdp</td>
<td>VSX Scalar Quad-Precision Compare Exponents</td>
</tr>
<tr>
<td>xscmphqpd</td>
<td>VSX Scalar Quad-Precision Compare Ordered</td>
</tr>
<tr>
<td>xscpsgnqp</td>
<td>VSX Scalar Quad-Precision CopySign</td>
</tr>
<tr>
<td>xscvdpq[p]</td>
<td>VSX Scalar Quad-Precision Convert From Double-Precision</td>
</tr>
<tr>
<td>xscvpdp[p]</td>
<td>VSX Scalar round &amp; Convert Quad-Precision to Double-Precision [using round to Odd]</td>
</tr>
<tr>
<td>xscvpqsdz</td>
<td>VSX Scalar truncate &amp; Convert Quad-Precision to Signed Doubleword</td>
</tr>
<tr>
<td>xscvpqswz</td>
<td>VSX Scalar truncate &amp; Convert Quad-Precision to Signed Word</td>
</tr>
<tr>
<td>xscvpqudz</td>
<td>VSX Scalar truncate &amp; Convert Quad-Precision to Unsigned Doubleword</td>
</tr>
<tr>
<td>xscvpquwz</td>
<td>VSX Scalar truncate &amp; Convert Quad-Precision to Unsigned Word</td>
</tr>
<tr>
<td>xscvqdqp</td>
<td>VSX Scalar Convert Signed Doubleword format to Quad-Precision format</td>
</tr>
<tr>
<td>xscvsqhp</td>
<td>VSX Scalar round &amp; Convert Double-Precision to Half-Precision</td>
</tr>
<tr>
<td>xscvqdp</td>
<td>VSX Scalar Convert Signed Doubleword format to Quad-Precision format</td>
</tr>
<tr>
<td>xsdivq[p]</td>
<td>VSX Scalar Quad-Precision Divide [&amp; round to Odd]</td>
</tr>
</tbody>
</table>

Table 3: Instructions controlled by the v2.07 bit (Sheet 3 of 6)
2.6 Other Hypervisor Resources

In addition to the resources described in the preceding sections, all hypervisor privileged instructions as well as the following resources are hypervisor resources, accessible to software only when the thread is in hypervisor state except as noted below.

- All implementation-specific resources except for privileged non-hypervisor implementation-specific SPRs. (See Section 5.4.4 for the list of the implementation-specific SPRs that are allowed to be privileged non-hypervisor SPRs.) Implementation-specific registers include registers (e.g., “HID” registers) that control hardware functions or affect the results of instruction execution. Examples include resources that disable caches, disable hardware error detection, set breakpoints, control power management, or significantly affect performance.

- ME bit of the MSR

- SPRs defined as hypervisor-privileged in Section 5.4.4. (Note: Although the Time Base, the PURR, and the SPURR can be altered only by a hypervisor program, the Time Base can be read by all programs and the PURR and SPURR can be read when the thread is in privileged state.)

The contents of a hypervisor resource can be modified by the execution of an instruction (e.g., `mtspr`) only in hypervisor state (MSR\textsubscript{HV \text{PR}} = 0b10). An attempt to modify the contents of a given hypervisor resource, other than MSR\textsubscript{ME}, in privileged but non-hypervisor state (MSR\textsubscript{HV \text{PR}} = 0b00) causes a Privileged Instruction type Program Interrupt when LPCR\textsubscript{EVIRT}=0 and a Privileged Instruction type Program Interrupt when LPCR\textsubscript{EVIRT}=0 and a

### Table 3: Instructions controlled by the v2.07 bit (Sheet 5 of 6)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsiexpdp</td>
<td>VSX Scalar Double-Precision Insert Exponent</td>
</tr>
<tr>
<td>xsiexpqp</td>
<td>VSX Scalar Quad-Precision Insert Exponent</td>
</tr>
<tr>
<td>xsmaddqp[o]</td>
<td>VSX Scalar Quad-Precision Multiply-Add [&amp; round to Odd]</td>
</tr>
<tr>
<td>xsmsubqp[o]</td>
<td>VSX Scalar Quad-Precision Multiply-Subtract [&amp; round to Odd]</td>
</tr>
<tr>
<td>xsnabsqp</td>
<td>VSX Scalar Quad-Precision Negative Absolute</td>
</tr>
<tr>
<td>xsnegqp</td>
<td>VSX Scalar Quad-Precision Negative</td>
</tr>
<tr>
<td>xsnmaddqp[o]</td>
<td>VSX Scalar Quad-Precision Negative Multiply-Add [&amp; round to Odd]</td>
</tr>
<tr>
<td>xsnmsubqp[o]</td>
<td>VSX Scalar Quad-Precision Negative Multiply-Subtract [&amp; round to Odd]</td>
</tr>
<tr>
<td>xsrqpi</td>
<td>VSX Scalar Round to Quad-Precision Integer</td>
</tr>
<tr>
<td>xsrqpix</td>
<td>VSX Scalar Round to Quad-Precision Integer with Inexact</td>
</tr>
<tr>
<td>xssqrtqsp[o]</td>
<td>VSX Scalar Quad-Precision Square Root [&amp; round to Odd]</td>
</tr>
<tr>
<td>xssubqsp[o]</td>
<td>VSX Scalar Quad-Precision Subtract [&amp; round to Odd]</td>
</tr>
<tr>
<td>xststdcqp</td>
<td>VSX Scalar Double-Precision Test Data Class</td>
</tr>
<tr>
<td>xststdcqp</td>
<td>VSX Scalar Quad-Precision Test Data Class</td>
</tr>
<tr>
<td>xststdcsp</td>
<td>VSX Scalar Single-Precision Test Data Class</td>
</tr>
<tr>
<td>xsexexpdp</td>
<td>VSX Scalar Double-Precision Extract Exponent</td>
</tr>
<tr>
<td>xsexexpqp</td>
<td>VSX Scalar Quad-Precision Extract Exponent</td>
</tr>
<tr>
<td>xxsigdp</td>
<td>VSX Scalar Double-Precision Extract Significand</td>
</tr>
<tr>
<td>xxsigqp</td>
<td>VSX Scalar Quad-Precision Extract Significand</td>
</tr>
<tr>
<td>xxvchpsp</td>
<td>VSX Vector Convert Half-Precision to Single-Precision</td>
</tr>
<tr>
<td>xxvsphp</td>
<td>VSX Vector round &amp; Convert Single-Precision to Half-Precision</td>
</tr>
<tr>
<td>xviexpdp</td>
<td>VSX Vector Double-Precision Insert Exponent</td>
</tr>
<tr>
<td>xviexpqp</td>
<td>VSX Vector Single-Precision Insert Exponent</td>
</tr>
<tr>
<td>xvtstdcqp</td>
<td>VSX Vector Double-Precision Test Data Class</td>
</tr>
<tr>
<td>xvtstdcsp</td>
<td>VSX Vector Single-Precision Test Data Class</td>
</tr>
</tbody>
</table>

### Table 3: Instructions controlled by the v2.07 bit (Sheet 6 of 6)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvxexpdp</td>
<td>VSX Vector Double-Precision Extract Exponent</td>
</tr>
<tr>
<td>xvxexpqp</td>
<td>VSX Vector Single-Precision Extract Exponent</td>
</tr>
<tr>
<td>xvxsigdp</td>
<td>VSX Vector Double-Precision Extract Significand</td>
</tr>
<tr>
<td>xvxsigqp</td>
<td>VSX Vector Single-Precision Extract Significand</td>
</tr>
<tr>
<td>xxbrd</td>
<td>VSX Vector Byte-Reverse Doubleword</td>
</tr>
<tr>
<td>xxbrh</td>
<td>VSX Vector Byte-Reverse Halfword</td>
</tr>
<tr>
<td>xxbrq</td>
<td>VSX Vector Byte-Reverse Quadword</td>
</tr>
<tr>
<td>xxbw</td>
<td>VSX Vector Byte-Reverse Word</td>
</tr>
<tr>
<td>xxextractuw</td>
<td>VSX Vector Extract Unsigned Word</td>
</tr>
<tr>
<td>xxinsertw</td>
<td>VSX Vector Insert Word</td>
</tr>
<tr>
<td>xxperm</td>
<td>VSX Vector Permute</td>
</tr>
<tr>
<td>xxpermr</td>
<td>VSX Vector Permute Right-indexed</td>
</tr>
<tr>
<td>xxsplitib</td>
<td>VSX Vector Splat Immediate Byte</td>
</tr>
</tbody>
</table>

Table 3: Instructions controlled by the v2.07 bit (Sheet 5 of 6)
Hypervisor Emulation Assistance interrupt when LPCREVIRT=1. An attempt to modify MSRME in privileged but non-hypervisor state is ignored (i.e., the bit is not changed).

--- Programming Note ---
Because the SPRs listed above are privileged for writing, an attempt to modify the contents of any of these SPRs in problem state (MSRPR=1) using mtspr causes a Privileged Instruction type Program exception, and similarly for MSRME.

### 2.7 Sharing Hypervisor and Ultravisor Resources

Shared SPRs are SPRs that are accessible to multiple threads. Changes to shared SPRs made by one thread are immediately readable (using mfspr) by all other threads sharing the SPR.

The LPIDR and DPDES must appear to software to be shared among threads of a sub-processor (see Section 2.8). If the implementation does not support sub-processors, the LPIDR and DPDES must be shared among all threads of the multi-threaded processor.

Certain additional hypervisor and ultravisor resources, and the PVR, may be shared among threads. Programs that modify these resources must be aware of this sharing, and must allow for the fact that changes to these resources may affect more than one thread.

The following additional resources may be shared among threads.
- HRMOR (see Section 2.3)
- LPIDR (see Section 2.4)
- PCR (see Section 2.5)
- URMOR (see Section 3.2)
- PVR (see Section 5.3.1)
- RPR (see Section 5.3.7)
- PTCR (see Section 6.7.6.1)
- AMOR (see Section 6.7.13.1)
- HMEER (see Section 7.2.11)
- Time Base (see Section 8.2)
- Virtual Time Base (see Section 8.3)
- Hypervisor Decrementer (see Section 8.5)
- certain implementation-specific registers or implementation-specific fields in architected registers

The set of resources that are shared is implementation-dependent.

Threads that share any of the resources listed above, with the exception of the PTCR, the PVR, the URMOR, and the HRMOR, must be in the same partition.

For each field of the LPCR, except the AIL, EVIRT, ONL, HDICE, MER, PECE, HEIC, and HVICE fields, software must ensure that the contents of the field are identical among all threads that are in the same partition and are not in hypervisor state.

Software must ensure that the contents of UILE and SMFCTRLE are identical among all threads in the system that have completed ultravisor initialization. The contents of the D and UDEE fields of SMFCTRL may differ among threads.

### 2.8 Sub-Processors

Hardware is allowed to sub-divide a multi-threaded processor into "sub-processors" that appear to privileged programs as multi-threaded processors with fewer threads. Such a multi-threaded processor appears to the hypervisor as a processor with a number of threads equal to the sum of all sub-processor threads, and in which the LPIDR for each sub-processor must appear to be shared among all threads of that sub-processor.

### 2.9 Thread Identification Register (TIR)

The TIR is a 64-bit read-only register that contains the thread number, which is a binary number corresponding to the thread.

For implementations that do not support sub-processors, the thread number of a thread is unique among all thread numbers of threads on the multi-threaded processor.

For implementations that support sub-processors, the value of this register depends on whether it is read in hypervisor or privileged, non-hypervisor state as follows.

- When this register is read in privileged, non-hypervisor state, the thread number is unique among all thread numbers of threads on the sub-processor.
- When this register is read in hypervisor state, the thread number is unique among all thread numbers of threads on the multi-threaded processor.

Threads are numbered sequentially, with valid values ranging from 0 to t-1, where t is the number of threads implemented. A thread for which TIR = n is referred to as "thread n."

The layout of the TIR is shown below.

<table>
<thead>
<tr>
<th>TIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

--- Figure 4. Thread Identification Register ---
Access to the TIR is privileged.
Since the thread number contained in this register is different if it is read in hypervisor state from when it is read in privileged, non-hypervisor state in implementations that support sub-processors, the following conventions are used.

- The value returned in privileged, non-hypervisor state is referred to as the “privileged thread number.”
- The value returned in hypervisor state is referred to as the “hypervisor thread number.”

### 2.10 Hypervisor Interrupt Little-Endian (HILE) Bit

The Hypervisor Interrupt Little-Endian (HILE) bit is a bit in an implementation-dependent register or similar mechanism. The contents of the HILE bit are copied into MSRLE by interrupts that result in MSR$_{HV}$ being equal to 0b01 (see Section 7.5), to establish the Endian mode for the interrupt handler. The HILE bit is set, by an implementation-dependent method, only during system initialization.

The contents of the HILE bit must be the same for all threads under the control of a given instance of the hypervisor; otherwise all results are undefined.
Chapter 3. Ultravisor and Secure Memory Facility (SMF)

3.1 Overview

The Secure Memory Facility (SMF) provides secure isolation of partitions from one another and from higher privilege system software. SMF functionality is implemented using a combination of hardware facilities and firmware that runs at a privilege level above the hypervisor. SMF targets a threat model in which the hypervisor can be compromised such that its inherent isolation capabilities can no longer be counted on. Maintaining the security of data is the sole objective of the ultravisor. It has no role in platform management and is not expected to deal with denial of service attacks. References elsewhere in the Books to “secure systems” apply more generally, and do not necessarily imply that the system uses SMF.

The SMF protection mechanism is based on the assignment of partitions to security domains. The hypervisor is in one security domain, along with all processes that run directly under the hypervisor and all partitions that do not take advantage of the SMF security capabilities. Each of the secure partitions is assigned to its own security domain so that its data and instructions can be protected from access by other security domains. A partition is identified as secure when MSRS=1. Each location in main storage has an associated Secure Memory property, memSM. Memory with memSM=1 may be referred to as “secure memory.” Memory with memSM=0 may be referred to as “ordinary memory.” The granularity and method with which main storage is mapped for the Secure Memory property is implementation specific. The Secure Memory property is commonly cached in the TLB and in implementation-specific looksaside buffers. When secure data are to be shared with untrusted software, the standard synchronization associated with PTE updates is used to regulate access. For example, prior to sharing secure data, the PTEs used to access the data are marked invalid and the corresponding TLB entries invalidated by the ultravisor using the standard invalidation sequence. (See Section 6.10.1.2.) The data are then encrypted and made available in ordinary memory (either memSM is turned off or the data are moved to ordinary memory). Finally the PTEs that will be used to access the data in ordinary memory are marked valid. (The last step may be done lazily.) Software running with MSR_S=0 is prohibited from accessing secure memory. Software running with MSR_S=1 may access both secure and ordinary memory.

SMF firmware runs in ultravisor state, a privilege level above that of the hypervisor. That firmware, along with the SMF hardware, is responsible for maintaining isolation of secure partitions from each other and from the hypervisor. This is accomplished by direct ultravisor management of the partition-scoped translation tables in secure memory for secure partitions. The ultravisor itself runs only in (ultravisor) real addressing mode. Security is the result of proper management of the partition-scoped translation together with the hardware enforcement of the access restriction for secure memory. With this hybrid approach, firmware has the ability to enable secure memory sharing between secure partitions and ordinary memory sharing between a given secure partition and the hypervisor, e.g. for system calls. The ultravisor can access any architecture resource or facility.

The hypervisor is expected to cooperate in the management of secure partitions by using ultravisor calls to dispatch them and to manage their storage allocations. To protect against programming errors and malicious hypervisor behavior, mtmsr[d], rfid, hrfd, and rtscv preserve MSR_S and hypervisor interrupts from secure partitions are always received in ultravisor state.

The purpose of intercepting hypervisor interrupts is to protect the state of the secure partition from the hypervisor. The ultravisor’s interrupt handler provides a ‘shim’ that saves and clears the processing state, and then transfers control to the hypervisor to handle the exception condition itself. The ultravisor will restore the secure partition state when it services the ultravisor call to (re-) dispatch the secure partition. Note that the ultravisor’s goal is merely to protect the security of
data, and not to provide broader system management oversight.

### 3.2 Ultravisor Real Mode Offset Register (URMOR)

The layout of the Ultravisor Real Mode Offset Register (URMOR) is shown in Figure 5 below.

**Figure 5. Ultravisor Real Mode Offset Register**

All other fields are reserved.

The supported URMO values are the non-negative multiples of $2^r$, where $r$ is the same implementation-dependent value that constrains the HRMO field of the HRMOR.

The contents of the URMOR affect how some storage accesses are performed as described in Sections 6.7.3 and 6.7.5.

### 3.3 Ultravisor Interrupt Little-Endian (UILE) Bit

The Ultravisor Interrupt Little-Endian (UILE) bit is a bit in an implementation-dependent register or similar mechanism. The contents of the UILE bit are copied into SMSCF by interrupts that result in MSR_S_HV being equal to 0b11 (see Section 7.5), to establish the Endian mode for the interrupt handler. The UILE bit is set, by an implementation-dependent method, only during system initialization.

The contents of the UILE bit must be the same for all threads in the system; otherwise all results are undefined.

### 3.4 Secure Memory Facility Control Register (SMFCTRL)

The Secure Memory Facility Control Register (SMFCTRL) is shown in Figure 6 below.

**Figure 6. Secure Memory Facility Control Register (SMFCTRL)**

If SMFCTRL_E=0, SMFCTRL_D and SMFCTRL_UDEE must be set to zero. References to SMFCTRL_D=1 or SMFCTRL_UDEE=1 elsewhere in the architecture assume SMFCTRL_E=1 unless otherwise stated or obvious from context.
3.4.1 Enabling SMF and Secure Memory Enforcement

The SMFCTRL_E bit enables SMF functionality. When SMFCTRL_E=1, certain facilities are ultravisor resources instead of hypervisor resources and secure memory checking is enabled.

Independent of the basic feature enablement above, SMF has state transition rules that facilitate the protection of security domains. (While these rules are nominally independent of the value of SMFCTRL_E, some transitions cannot happen when SMFCTRL_E=0. Specifically, ultravisor interrupts cannot occur when SMFCTRL_E=0.)

- All interrupts that are not ultravisor interrupts preserve MSR_S. (Ultravisor interrupts necessarily set MSR_S to 1.)
- mtmsr[d], rfid, hrfid, and rfscv are not permitted to change MSR_S.

Table 4 summarizes the effect of the SMFCTRL_E bit and the MSR_S.HV.PR bits on various facilities.
<table>
<thead>
<tr>
<th>facility</th>
<th>MSR&lt;sub&gt;S&lt;/sub&gt; HV/PR</th>
<th>SMFCTRL&lt;sub&gt;E&lt;/sub&gt;</th>
<th>LPCR&lt;sub&gt;EVIRT&lt;/sub&gt;</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mtspr</strong> or <strong>mfspr</strong> specifying URMOR, USRR0, USRR1, USPRG0, USPRG1, or SMFCTRL; urfid, msgsndu, msgclru</td>
<td>110</td>
<td>dc</td>
<td>dc</td>
<td>execution allowed</td>
</tr>
<tr>
<td></td>
<td>all xxx except 110**</td>
<td>dc</td>
<td>dc</td>
<td>Privileged Instruction type Program interrupt to xx0</td>
</tr>
<tr>
<td><strong>mtspr</strong> specifying PTCR</td>
<td>110</td>
<td>dc</td>
<td>dc</td>
<td>execution allowed</td>
</tr>
<tr>
<td></td>
<td>010</td>
<td>0</td>
<td>dc</td>
<td>execution allowed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>dc</td>
<td>HEAI to 010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x00</td>
<td>dc</td>
<td>0</td>
<td>Privileged Instruction type Program interrupt to x00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>HEAI to x10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xx1**</td>
<td>dc</td>
<td>dc</td>
<td>Privileged Instruction type Program interrupt to xx0</td>
</tr>
<tr>
<td><strong>mtspr</strong> or <strong>mfspr</strong> specifying DAWRn, DAWRXn or CIABR when SMFCTRL&lt;sub&gt;D&lt;/sub&gt;=1</td>
<td>110</td>
<td>1</td>
<td>dc</td>
<td>execution allowed</td>
</tr>
<tr>
<td></td>
<td>010</td>
<td>1</td>
<td>dc</td>
<td>HEAI to 010</td>
</tr>
<tr>
<td></td>
<td>x00</td>
<td>1</td>
<td>0</td>
<td>Privileged Instruction type Program interrupt to x00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>HEAI to x10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xx1**</td>
<td>1</td>
<td>dc</td>
<td>Privileged Instruction type Program interrupt to xx0</td>
</tr>
<tr>
<td><strong>sc</strong> 2 instruction</td>
<td>dc**</td>
<td>0</td>
<td>dc</td>
<td>hypervisor call, but with SRR1 showing LEV=2</td>
</tr>
<tr>
<td></td>
<td>dc**</td>
<td>1</td>
<td>dc</td>
<td>ultravisor call</td>
</tr>
<tr>
<td><strong>mem&lt;sub&gt;SM&lt;/sub&gt; evaluation and match</strong></td>
<td>dc**</td>
<td>0</td>
<td>dc</td>
<td>disabled</td>
</tr>
<tr>
<td></td>
<td>dc**</td>
<td>1</td>
<td>dc</td>
<td>enabled*</td>
</tr>
</tbody>
</table>

* mem<sub>SM</sub> evaluation may be avoided when MSR<sub>S</sub>=1, depending on translation cache design
dc = don't care
** The encoding MSR<sub>S</sub> HV/PR=0b111 is reserved and must not be used.

Table 4: Ultravisor Resource Behavior

---

**Programming Note**

Access to memory by mechanisms outside the core must also enforce secure memory access restrictions. Facilities that translate addresses or otherwise use real addresses to access memory must check mem<sub>SM</sub> against PATE<sub>S</sub> for the partition on behalf of which they access memory.

Such mechanisms will require a means to evaluate mem<sub>SM</sub> and a proxy for SMFCTRL<sub>E</sub> to provide the same enablement function for secure memory access enforcement as in the core.

In addition or as an alternative, TCE tables may be managed by the ultravisor and used to identify regions of memory that I/O devices may access.
Chapter 4. Branch Facility

4.1 Branch Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Facility that are not covered in Book I.

4.2 Branch Facility Registers

4.2.1 Machine State Register

The Machine State Register (MSR) is a 64-bit register. This register defines the state of the thread. On interrupt, the MSR bits are altered in accordance with Figure 67 on page 1261. The MSR can also be modified by the \texttt{mtmsr[d]}, \texttt{rfscv}, \texttt{rfid}, \texttt{hrfid}, and \texttt{urfid} instructions. It can be read by the \texttt{mfmsr} instruction.

**Figure 7. Machine State Register**

Below are shown the bit definitions for the Machine State Register.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sixty-Four-Bit Mode (SF)</td>
</tr>
<tr>
<td>0</td>
<td>The thread is in 32-bit mode.</td>
</tr>
<tr>
<td>1</td>
<td>The thread is in 64-bit mode.</td>
</tr>
<tr>
<td></td>
<td>Software must ensure that SF=1 whenever the thread is in ultravisor state.</td>
</tr>
<tr>
<td>1:2</td>
<td>Reserved</td>
</tr>
<tr>
<td>3</td>
<td>Hypervisor State (HV)</td>
</tr>
<tr>
<td>0</td>
<td>The thread is not in hypervisor state.</td>
</tr>
<tr>
<td>1</td>
<td>If MSR\textsubscript{PR}=0, the thread is in hypervisor state; otherwise the thread is not in hypervisor state.</td>
</tr>
</tbody>
</table>

**Programming Note**

The privilege state of the thread is determined by MSR\textsubscript{S}, MSR\textsubscript{HV}, and MSR\textsubscript{PR}, as follows.

<table>
<thead>
<tr>
<th>S</th>
<th>HV</th>
<th>PR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>1</td>
<td>problem</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>problem</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>0</td>
<td>privileged</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>0</td>
<td>hypervisor</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ultravisor</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>reserved</td>
</tr>
</tbody>
</table>

Hypervisor state is also a privileged state (MSR\textsubscript{PR} = 0). All references to “privileged state” in the Books include hypervisor state unless otherwise stated or obvious from context. Ultravisor state is also a hypervisor state (MSR\textsubscript{HV} PR = 0b10). All references to “hypervisor state” in the Books include ultravisor state unless otherwise stated or obvious from context.

MSR\textsubscript{HV} can be set to 1 only by the \texttt{System Call} instruction and some interrupts. It can be set to 0 only by \texttt{rfid}, \texttt{hrfid}, and \texttt{urfid}.

It is possible to run an operating system in an environment that lacks a hypervisor, by always having MSR\textsubscript{HV} = 1 and using MSR\textsubscript{HV} PR = 0b10 for the operating system (effectively, the OS runs in hypervisor state) and MSR\textsubscript{HV} PR = 0b11 for applications. In this use, MSR\textsubscript{S} would be 0, and the environment would also lack an ultravisor.

Reserved

Software must ensure that this bit contains 0; otherwise the results of executing all instructions are boundedly undefined.
This bit is initialized to 0 by hardware at system bringup. The handling of this bit by interrupts and by the `rfid`, `hrfid`, `urfid`, and `rsfscv` instructions is such that, unless software deliberately sets the bit to 1, the bit will continue to contain 0.

### Vector Available (VEC)

0 The thread cannot execute any vector instructions, including vector loads, stores, and moves.
1 The thread can execute vector instructions unless they have been made unavailable by some other register.

### Problem State (PR)

0 The thread is in privileged state.
1 If MSR$_{HVPR} = 0b11$, the thread is in problem state.

### Programming Note

An application binary interface defined to support Vector-Scalar operations should also specify a requirement that MSR$_{FP}$ and MSR$_{VEC}$ be set to 1 whenever MSR$_{VSX}$ is set to 1.

### Secure (S)

0 The thread is not in Secure state. It may not access Secure memory. The thread is not in ultravisor state.
1 The thread is in Secure state. If MSR$_{HV}=1$ and MSR$_{PR}=0$, the thread is in ultravisor state; otherwise the value does not affect privilege. The state with MSR$_{HV}=1$ and MSR$_{PR}=1$ is reserved. Software must not set MSR$_{HVPR} = 0b111$. References elsewhere in this document to MSR$_{HVPR}=0b11$ assume MSR$_{S}=0$ unless otherwise stated or obvious from context.

### Machine Check Interrupt Enable (ME)

0 Machine Check interrupts are disabled.
1 Machine Check interrupts are enabled.

This bit is a hypervisor resource; see Chapter 2, “Logical Partitioning (LPAR) and Thread Control”, on page 1121.

### Programming Note

The only instructions that can alter MSR$_{ME}$ are `rfid`, `hrfid`, and `urfid`.

### Floating-Point Available (FP)

0 The thread cannot execute any floating-point instructions, including floating-point loads, stores, and moves.
1 The thread can execute floating-point instructions unless they have been made unavailable by some other register.

### Programming Note

Any instruction that sets MSR$_{PR}$ to 1 also sets MSR$_{EE}$, MSR$_{IR}$, and MSR$_{DR}$ to 1.

The state with MSR$_{HVPR} = 0b111$ is reserved.

### Floating-Point Exception Mode 0 (FE0)

See below.

### Trace Enable (TE)

00 Trace Disabled: The thread executes instructions normally.
01 Branch Trace: The thread generates a Branch type Trace interrupt after completing the execution of a branch instruction, whether or not the branch is taken.

10 Single Step Trace: The thread generates a Single-Step type Trace interrupt after successfully completing the execution of the next instruction, unless that instruction is a \textit{urfid}, \textit{hrfid}, \textit{rfid}, \textit{rfscv}, or a Power-Saving Mode instruction, all of which are never traced. Successful completion means that the instruction caused no other interrupt.

11 Reserved.

Branch tracing need not be supported. If the function is not implemented, the 0b01 bit encoding is treated as reserved.

55 \textbf{Floating-Point Exception Mode 1 (FE1)}

See below.

56:57 Reserved

58 \textbf{Instruction Relocate (IR)}

0 Instruction address translation is disabled.
1 Instruction address translation is enabled.

\textbf{Programming Note}

See the Programming Note in the definition of MSR\textsubscript{S} and in the definition of MSR\textsubscript{PR}.

59 \textbf{Data Relocate (DR)}

0 Data address translation is disabled. Effective Address Overflow (EAO) (see Book I) does not occur.
1 Data address translation is enabled. EAO causes a Data Storage interrupt.

60 Reserved

61 \textbf{Performance Monitor Mark (PMM)}

This bit is used by software in conjunction with the Performance Monitor, as described in Chapter 10.

Before hypervisor support was added to the architecture, “translation is disabled” for MSR\textsubscript{IR}=0 truly meant that no translation was performed for instruction addresses, and correspondingly for MSR\textsubscript{DR}=0 for data addresses. The architecture continues to use “translation is disabled” to refer to MSR\textsubscript{IR}=0 and MSR\textsubscript{DR}=0 despite that the behavior today is more complicated. When MSR\textsubscript{HV IR}=0b10, it is still true that no translation is performed for instruction addresses, and correspondingly for data addresses if MSR\textsubscript{HV DR}=0b10. But in privileged non-hypervisor state when MSR\textsubscript{IR}=0 or MSR\textsubscript{DR}=0, limited translation is performed under control of the hypervisor. For an HPT translation guest, translation is performed as described in Section 6.7.3.3, with storage exceptions directed to the hypervisor. For a Radix Tree Translation guest, only partition-scoped translation is performed, with storage exceptions directed to the hypervisor.
0 The thread is in Big-Endian mode.
1 The thread is in Little-Endian mode.

Programming Note

The only instructions that can alter MSR LE are rfd, hrdid, urfd, and rfscv.

The Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below. For further details see Book I.

<table>
<thead>
<tr>
<th>FE0</th>
<th>FE1</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Ignore Exceptions</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Imprecise Nonrecoverable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Imprecise Recoverable</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Precise</td>
</tr>
</tbody>
</table>

Recoverable Interrupt (RI)

0 Interrupt is not recoverable.
1 Interrupt is recoverable.

Additional information about the use of this bit is given in Sections 7.4.3, “Interrupt Processing” on page 1256, 7.5.1, “System Reset Interrupt” on page 1262, and 7.5.2, “Machine Check Interrupt” on page 1264.

Little-Endian Mode (LE)

Common uses of the PMM bit include the following.

- All counters count events for a few selected processes. This use requires the following bit settings:
  - MSR\textsubscript{PMM} = 1 for the selected processes, MSR\textsubscript{PMM} = 0 for all other processes
  - MMCR\textsubscript{FCM0} = 1
  - MMCR\textsubscript{FCM1} = 0
  - MMCR2 = 0x0000

- All counters count events for all but a few selected processes. This use requires the following bit settings:
  - MSR\textsubscript{PMM} = 1 for the selected processes, MSR\textsubscript{PMM} = 0 for all other processes
  - MMCR\textsubscript{FCM0} = 0
  - MMCR\textsubscript{FCM1} = 1
  - MMCR2 = 0x0000

Notice that for both of these uses a mark value of 1 identifies the “few” processes and a mark value of 0 identifies the remaining “many” processes. Because the PMM bit is set to 0 when an interrupt occurs (see Figure 67 on page 1261), interrupt handlers are treated as one of the “many”. If it is desired to treat interrupt handlers as one of the “few”, the mark value convention just described would be reversed.

If only a specific counter n is to be frozen, MMCR\textsubscript{FCM0} FCM\textsubscript{1} is set to 0b00, and MMCR\textsubscript{FCM0} FCM\textsubscript{1} instead of MMCR\textsubscript{FCM0} FCM\textsubscript{1} and MMCR\textsubscript{FCM1} FCM\textsubscript{1} are set to the values described above.
4.2.2 Processor Stop Status and Control Register (PSSCR)

The layout of the PSSCR is shown below.

```
|   PLS |   /   /   |   0  |  4  |  8  | 12  | 16  | 20  | 24  | 28  | 32  | 36  | 40  | 44  | 48  | 52  | 56  | 60  |
|-------|----------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
```

**Figure 8. Processor stop Status and Control Register**

The contents of the PSSCR control the operation of the `stop` instruction and provide status indicating the level of power saving that was entered while in power-saving mode.

All fields of this register can be read and written by the hypervisor using either hypervisor SPR 855 or privileged SPR 823. A subset of the fields of this register can be read and written in privileged non-hypervisor state using privileged SPR 823, as specified below. Fields that can only be read or written by the hypervisor are indicated below; all other fields can be read or written in either privileged non-hypervisor or hypervisor states. When a field that is accessible only to the hypervisor is accessed in privileged non-hypervisor state, writes have no effect and reads return 0s regardless of the value of the field.

The bits and their meanings are as follows.

- **0:3 Power-Saving Level Status (PLS)**
  
  Hardware sets this field to the highest power-saving level that the thread entered between the time when the `stop` instruction is executed and when the thread exits power-saving mode. See the description of the SD field for the value returned in this field when the PSSCR is read.

  **Programming Note**

  Since the power-saving level entered during power-saving mode may vary with time, the PLS field may not indicate the power-saving level that existed at exit from power-saving mode.

- **4:40 Reserved**

- **41 Status Disable (SD)**

  This field is accessible only to the hypervisor.

  0 The current value of the PLS field is returned in the PLS field when reading the PSSCR (using `mfspr`).

  1 0’s are returned in the PLS field when reading the PSSCR (using `mfspr`).

  ```
  Programming Note
  ```

  Exception if SMFCTRLUDEE was set to 0 when

  ```

  **Enable State Loss (ESL)**

  This field is accessible only to the hypervisor.

  0 State loss while in power-saving mode is controlled by the RL, MTL, and PSLL fields.

  1 Non-hypervisor state loss is allowed while in power-saving mode in addition to state loss controlled by the RL, MTL, and PSLL fields.

  If this field is set to 1 when the `stop` instruction is executed in privileged non-hypervisor state, a Hypervisor Facility Unavailable interrupt occurs. See Section 7.5.26.

  For power-saving levels that allow loss of the LPCR, implementations must provide the means to exit power-saving mode upon the occurrence of a System Reset exception and any of the exceptions that were enabled by the PECE field when the `stop` instruction was executed. For this case, the implementation is also allowed to exit on the occurrence of any exceptions that were disabled by the PECE as well.

  For power-saving levels that allow loss of SMFCTRL, implementations must provide the means to exit power-saving mode upon the occurrence of a Directed Ultravisor Doorbell exception if SMFCTRLUDEE was set to 1 when `stop` was executed. For this case, the implementation is also allowed to exit on the occurrence of a Directed Ultravisor Doorbell `stop` was executed.
Exit Criterion (EC)

This field is accessible only to the hypervisor.

0 Hardware will exit power-saving mode when the exception corresponding to any system-caused interrupt occurs. Power-saving mode is exited either at the instruction following the stop (if MSR_{EE}=0) or in the corresponding interrupt handler (if MSR_{EE}=1).

1 If SMFCtrl_{UDED} was set to 1 when stop was executed and SMFCtrl_{UDED} was not lost, hardware will exit power-saving mode when a Directed Ultravisor Doorbell exception occurs. If LPCR_{PECE} is not lost, hardware will exit power-saving mode when a System Reset exception or one of the events specified in LPCR_{PECE} occurs. If the event is a Machine Check exception, then a Machine Check interrupt occurs; otherwise a System Reset interrupt occurs, and the contents of SRR1 indicate the event that caused exit from power-saving mode.

When the stop instruction is executed in hypervisor state, the hypervisor must set the ESL field to the same value as this field. Also, if the RL or MTL fields are set to values that allow state loss, then fields ESL and EC must both be set to 1. Other combinations of the values of the ESL, EC, RL, and MTL fields are reserved for future use.

Power-Saving Level Limit (PSLL)

This field is accessible only to the hypervisor.

This field limits the power-saving level that may be entered or transitioned into when the stop instruction is executed in privileged non-hypervisor state; when the stop instruction is executed in hypervisor state, this field is ignored.

Transition Rate (TR)

This field is used to specify the relative rate at which the power-saving level increases during power-saving mode. The rate of power-saving level increase corresponding to each value is implementation-dependent, and monotonically increasing with the value specified.

Maximum Transition Level (MTL)

If the value of this field is greater than the value of the Power-Saving Level Limit (PSLL) field when stop is executed in privileged non-hypervisor state, a Hypervisor Facility Unavailable interrupt occurs. See Section 7.5.26 of Book III.

Otherwise, if the value of this field is greater than the value of the RL field, the power-saving level is allowed to increase from the value in the RL field up to the value of this field during power-saving mode.

If this field is less than or equal to the value of the PSLL field when stop is executed in privileged non-hypervisor state, this field is used to specify the maximum power-saving level that can be reached during power-saving mode provided that the value of this field is greater than the value of the RL field. If this field is less than the Requested Level (RL) field when stop is executed hardware is not allowed to increase the power-saving level beyond the value indicated in the RL field.

Requested Level (RL)

When state loss occurs, thread resources such as SPRs, GPRs, address translation resources, etc. may be powered off or allocated to other threads during power-saving mode. The amount of state loss for various combinations of ESL, RL, and MTL values is implementation dependent, subject to the restrictions specified in Section 4.3.2.

In order to enable an OS to enter power-saving mode without hypervisor involvement, both the EC and ESL bits must be set to 0s. When this is done, OS execution of the stop instruction will not cause hypervisor involvement provided that bits RL and MTL are less than or equal to PSLL. See Section 7.5.26 for details.

Programming Note

In order to enable an OS to enter power-saving mode without hypervisor involvement, both the EC and ESL bits must be set to 0s. When this is done, OS execution of the stop instruction will not cause hypervisor involvement provided that bits RL and MTL are less than or equal to PSLL. See Section 7.5.26 for details.

Programming Note

If the event is a Machine Check exception, then a Machine Check interrupt occurs; otherwise a System Reset interrupt occurs, and the contents of SRR1 indicate the event that caused exit from power-saving mode.

When the stop instruction is executed in hypervisor state, the hypervisor must set the ESL field to the same value as this field. Also, if the RL or MTL fields are set to values that allow state loss, then fields ESL and EC must both be set to 1. Other combinations of the values of the ESL, EC, RL, and MTL fields are reserved for future use.

Other combinations of the values of the ESL, EC, RL, and MTL fields may be allowed in a future version of the architecture in order to provide additional functionality.

If this field is set to 1 when the stop instruction is executed in privileged non-hypervisor state, a Hypervisor Facility Unavailable interrupt occurs. See Section 7.5.26.
This field is used to specify the power-saving level that is to be entered when the `stop` instruction is executed.

If the value of this field is greater than the value of the Power-Saving Level Limit (PSLL) field when `stop` is executed in privileged non-hypervisor state, a Hypervisor Facility Unavailable interrupt occurs.

---

**Programming Note**

The Hypervisor Facility Unavailable interrupt occurs when a privileged non-hypervisor program executes `stop` when PSSCR<sub>RL</sub> > PSSCR<sub>PSLL</sub> so that the Hypervisor may decide whether or not to allow the requested loss of state to occur.

If the hypervisor decides that some loss of state is acceptable, it may choose to re-execute `stop` after either setting PSSCR<sub>MTL</sub> to a value that causes state loss, or setting both PSSCR<sub>RL</sub> and PSSCR<sub>MTL</sub> to values that cause state loss. When the thread exits power-saving mode, the hypervisor can quickly determine whether any resources were actually lost and need to be restored.
4.3 Branch Facility Instructions

4.3.1 System Linkage Instructions

These instructions provide the means by which a program can call upon the system to perform a service, and by which the system can return from performing a service or from processing an interrupt.

The System Call instruction is described in Book I, but only at the level required by an application programmer. A complete description of this instruction appears below.

**System Call**

**SC-form**

```
    sc    LEV
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>20</th>
<th>27</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SRR0 ← lea CIA + 4
SRR1[33:36 42:47] ← 0
MSR ← new value (see below)
NIA ← 0x0000_0000_0000_0C00

The effective address of the instruction following the System Call instruction is placed into SRR0. Bits 0:32, 37:41, and 48:63 of the MSR are placed into the corresponding bits of SRR1, and bits 33:36 and 42:47 of SRR1 are set to zero.

Then a System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 7.5, "Interrupt Definitions" on page 1260. The setting of the MSR is affected by the contents of the LEV field. LEV values greater than 2 are reserved. Bits 0:4 of the LEV field (instruction bits 20:24) are treated as a reserved field.

The interrupt causes the next instruction to be fetched from effective address 0x0000_0000_0000_0C00.

This instruction is context synchronizing.

**Special Registers Altered:**

SRR0  SRR1  MSR

--- Programming Note

If LEV=1, the hypervisor is invoked.

If LEV=2 and SMFCTRL_E = 1, the ultravisor is invoked.

If LEV=2 and SMFCTRL_E = 0, the hypervisor is invoked. However, such invocation should be considered a programming error.

Executing this instruction with LEV=1 or LEV=2 is the only way that executing an instruction can cause a transition from non-hypervisor state to hypervisor state on the thread that executed the instruction. Executing this instruction with LEV=2 when SMFCTRL_E=1 is the only way that executing an instruction can cause a transition from non-ultravisor state to ultravisor state on the thread that executed the instruction.

In correct use, this instruction is used to "call up" one privilege level (application program calls operating system, operating system calls hypervisor, hypervisor calls ultravisor). However, it is possible for a program to call up more than one level (e.g., for an application program to call the hypervisor). An attempt to call up more than one level should be considered a programming error.

--- Programming Note

**sc** serves as both a basic and an extended mnemonic. The Assembler will recognize an **sc** mnemonic with one operand as the basic form, and an **sc** mnemonic with no operand as the extended form. In the extended form the LEV operand is omitted and assumed to be 0.
**System Call Vectored SC-form**

```
scv         LEV
```

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>20</td>
<td>27</td>
<td>30</td>
</tr>
</tbody>
</table>

- `LR` ← CIA + 4
- `CTR_{33:36 | 42:47}` ← undefined
- `CTR_{0:32 | 37:41}` ← MSR_{0:32 | 37:41}
- `MSR` ← new_value (see below)
- `NIA` ← (see below)

The effective address of the instruction following the `System Call Vectored` instruction is placed into the Link Register. Bits 0:32, 37:41, and 48:63 of the MSR are placed into the corresponding bits of Count Register, and bits 33:36 and 42:47 of Count Register are set to undefined values.

Then a System Call Vectored interrupt is generated. The interrupt causes the MSR to be altered as described in Section 7.5.

The interrupt causes the next instruction to be fetched as specified in LPCR_{AIL} or LPCR_{HAIL} as appropriate (see Section 2.2).

The SRRs are not affected.

This instruction is context synchronizing.

**Special Registers Altered:**

- LR
- CTR
- MSR

**Return From System Call Vectored XL-form**

```
rfscv
```

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

- `MSR_{48} ← CTR_{48} | CTR_{49}
- `MSR_{38} ← (CTR_{38} | CTR_{39})
- `MSR_{29} ← (MSR_{2} & MSR_{3} & (¬CTR_{3})
- `MSR_{24} ← (CTR_{23} | CTR_{24})
- `MSR_{0:2 | 37:40 | 49:50 | 52:57 | 60:63} ← CTR_{0:2 | 37:40 | 49:50 | 52:57 | 60:63 | LEV}
- `NIA ← lea LR_{0:61} | 0b00"

The result of ORing bits 48 and 49 of the Count Register is placed into MSR_{48}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of the Count Register is complemented and then ANDed with the result of ORing bits 58 and 49 of the Count Register and placed into MSR_{58}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of the Count Register is complemented and then ANDed with the result of ORing bits 58 and 49 of the Count Register and placed into MSR_{58}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of the Count Register is complemented and then ANDed with the result of ORing bits 58 and 49 of the Count Register and placed into MSR_{58}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of the Count Register is complemented and then ANDed with the result of ORing bits 58 and 49 of the Count Register and placed into MSR_{58}. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address LR_{0:61} | 0b00 (when SF=1 in the new MSR value) or LR_{0:61} | 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, HSRR0, or USRR0 by the interrupt processing mechanism (see Section 7.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing.

**Special Registers Altered:**

- MSR

---

**Programming Note**

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1. If this instruction results in MSR_{HV PR} being equal to 0b110, it also sets MSR_{IR} and MSR_{DR} to 0.

This instruction does not alter MSR_{HV}, MSR_{S}, or MSR_{ME}.
Return From Interrupt Doubleword

**XL-form**

rfid

<table>
<thead>
<tr>
<th>0</th>
<th>19</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>b</td>
<td>h1</td>
</tr>
</tbody>
</table>

MSR51 ← (MSR3 & SRR1:51) | (¬MSR3) & MSR51
MSR3 ← MSR3 & SRR13
MSR48 ← SRR1:48 | MSR1:49
MSR58 ← (CTR:58 | CTR:49)
& ¬(MSR41 & MSR3 & (¬CTR:49))
MSR59 ← (CTR:59 | CTR:49)
& ¬(MSR41 & MSR3 & (¬CTR:49))
NIA ← SRR0:0:61 | 0b00

If MSR3=1 then bits 3 and 51 of SRR1 are placed into the corresponding bits of the MSR. The result of ORing bits 48 and 49 of SRR1 is complemented and then ANDed with the result of ORing bits 58 and 49 of SRR1 placed into MSR58. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of SRR1 is complemented and then ANDed with the result of ORing bits 58 and 49 of SRR1 placed into MSR58. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of SRR1 is complemented and then ANDed with the result of ORing bits 59 and 49 of SRR1 placed into MSR59. Bits 0:2, 4:32, 37:40, 49:57, and 60:63 of SRR1 are placed into the corresponding bits of the MSR.

If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address SRR0:0:61 | 0b00 (when SF=1 in the new MSR value) or 320 | SRR0:32:61 | 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, HSRR0, or USRR0 by the interrupt processing mechanism (see Section 7.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing.

**Special Registers Altered:**

- MSR

---

**Programming Note**

If this instruction sets MSRPR to 1, it also sets MSREE, MSRIR, and MSRDR to 1. If this instruction results in MSRSHVPR being equal to 0b110, it also sets MSRIR and MSRDR to 0.

Hypervisor Return From Interrupt Doubleword

**XL-form**

hrfid

<table>
<thead>
<tr>
<th>0</th>
<th>19</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>b</td>
<td>h1</td>
</tr>
</tbody>
</table>

MSR48 ← HSRR1:48 | HSRR1:49
MSR58 ← (CTR:58 | CTR:49)
& ¬(MSR41 & MSR3 & (¬CTR:49))
MSR59 ← (CTR:59 | CTR:49)
& ¬(MSR41 & MSR3 & (¬CTR:49))
MSR0:0:2 37:40 49:57 60:63 ← HSRR1:0:2 37:40 49:57 60:63
NIA ← SRR0:0:61 | 0b00

The result of ORing bits 48 and 49 of HSRR1 is placed into MSR48. The result of ANDing bit 41 of the MSR with bit 3 of HSRR1 and with the complement of bit 49 of HSRR1 is complemented and then ANDed with the result of ORing bits 58 and 49 of HSRR1 placed into MSR58. The result of ANDing bit 41 of the MSR with bit 3 of HSRR1 and with the complement of bit 49 of HSRR1 is complemented and then ANDed with the result of ORing bits 59 and 49 of HSRR1 placed into MSR59. Bits 0:2, 37:40, 49:57, and 60:63 of HSRR1 are placed into the corresponding bits of the MSR.

If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address HSRR0:0:61 | 0b00 (when SF=1 in the new MSR value) or 320 | HSRR0:32:61 | 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, HSRR0, or USRR0 by the interrupt processing mechanism (see Section 7.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is hypervisor privileged and context synchronizing.

**Special Registers Altered:**

- MSR

---

**Programming Note**

If this instruction sets MSRPR to 1, it also sets MSREE, MSRIR, and MSRDR to 1. If this instruction results in MSRSHVPR being equal to 0b110, it also sets MSRIR and MSRDR to 0.
Ultravisor Return From Interrupt
Doubleword         XL-form

urfd

<table>
<thead>
<tr>
<th>19</th>
<th>///</th>
<th>///</th>
<th>///</th>
<th>306</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

|       | MSR48 ← USRR148 | USRR149  |
| 32    | MSR58 ← (USRR158 | USRR149) |
| 32    | & ¬(USRR141 & USRR13 & {¬USRR149}) |
| 32    | MSR59 ← (USRR159 | USRR149) |
| 32    | & ¬(USRR141 & USRR13 & {¬USRR149}) |
|       | NIA ← lseq USRR0.61 || 0b00 |

The result of ORing bits 48 and 49 of USRR1 is placed into MSR48. The result of ANDing bit 41 of USRR1 with bit 3 of USRR1 and with the complement of bit 49 of USRR1 is complemented and then ANDed with the result of ORing bits 58 and 49 of USRR1 and placed into MSR58. The result of ANDing bit 41 of USRR1 with bit 3 of USRR1 and with the complement of bit 49 of USRR1 is complemented and then ANDed with the result of ORing bits 59 and 49 of USRR1 and placed into MSR59. Bits 0:32, 37:41, 49:57, and 60:63 of USRR1 are placed into the corresponding bits of the MSR.

If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address USRR0.61 || 0b00 (when SF=1 in the new MSR value) or 32 || USRR0.32.61 || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, HSRR0, or USRR0 by the interrupt processing mechanism (see Section 7.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is ultravisor privileged and context synchronizing.

Special Registers Altered:
MSR

Programming Note

If this instruction sets MSRPR to 1, it also sets MSREE, MSRIR, and MSDR to 1. If this instruction sets MSRHVPR to 0b110, it also sets MSRIR and MSDR to 0.
4.3.2 Power-Saving Mode

Power-Saving Mode is a mode in which the thread does not execute instructions and may consume less power than it would if it were not in power-saving mode.

There are 16 levels of power savings, designated as levels 0-15. For each power-saving level, the power consumed may be less than or equal to the power consumed in the next-lower level, and the time required for the thread to exit power-saving mode and resume execution may be greater than or equal that of the next-lower level.

When the thread is in power-saving mode, some resource state may be lost. The state that may be lost while in each power-saving level is implementation dependent, with the following restrictions.

- For PSSCRESL = 0 and power-saving level 0000, no thread state is lost.
- There must be a power-saving level in which the Decrementer and all hypervisor resources are maintained as if the thread was not in power-saving mode, and in which sufficient information is maintained to allow the hypervisor to resume execution.
- The amount of state loss in a given level is less than or equal to the amount of state loss in the next higher level.
- The state of all read-only resources, SMFCTRL\_E, and the URMOR in an SMF-enabled system or the HRMOR in an SMF-disabled system is always maintained.

**Programming Note**

For the power-saving level corresponding to the second item above, if the state of the Decrementer were not maintained and updated as if the thread was not in power-saving mode, Decrementer exceptions would not reliably cause exit from this power-saving level even if Decrementer exceptions were enabled to cause exit.
4.3.2.1 Power-Saving Mode Instruction

The **stop** instruction is used to stop instruction fetching and execution and put the thread into power-saving mode. The thread remains in power-saving mode until a system reset exception or an event that is enabled to cause exit from power-saving mode occurs. (See the definition of PSSCREC in Section 4.2.2.)

### stop  Xl-form

```
stop
```

The thread is placed into power-saving mode and execution is stopped.

The power-saving level that is entered is determined by the contents of the PSSCR (see Section 4.2.2). The thread state that is maintained depends on the power-saving level that is entered. The thread state that is maintained at each power-saving level is implementation-dependent, subject to the restrictions specified in Section 4.3.2. If MSREE=0 or in the corresponding interrupt handler (if MSREE=1).

If MSREE=1 when the **stop** instruction is executed, then the interrupt corresponding to the exception that was expected to cause exit from power-saving mode may occur immediately prior to execution of the **stop** instruction. If this occurs, the result may be a software hang condition since the exception that was expected to cause exit from power-saving mode has already occurred.

The above software hang condition can be prevented by setting MSR_EE=0 prior to executing **stop**.

After the thread has entered power-saving mode with PSSCREC=0, any exception may cause exit from power-saving mode. When an exception occurs, power-saving mode is exited either at the instruction following the **stop** (if MSREE=0) or in the corresponding interrupt handler (if MSREE=1).

### Programming Note

If **stop** was executed when PSSCREC=0, then PSSCRESL must also be set to 0 and PSSCREL must be set to values that do not allow state loss. (See the definition of the EC bit description in Section 4.3.2.) This guarantees that the state of MSREE is not lost.

### Special Registers Altered:
None

4.3.2.2 Entering and Exiting Power-Saving Mode

Before software executes the **stop** instruction, the PSSCR is initialized. If the **stop** instruction is to be used by the OS, the hypervisor initializes the fields that are accessible only to the hypervisor before dispatching the OS. These fields include the SD, ESL, EC, and PSLL fields. See the Programming Notes for these fields in Section 4.2.2 for additional information.

If the **stop** instruction is to be executed by the hypervisor when PSSCREC=1, LPCR_PEC and SMFCTR_LUDEE must be set to the desired value (see Sections 2.2 and 3.4). Depending on the implementation and the power-saving level to be entered, it may also be necessary to save the state of certain resources and perform synchronization procedures to ensure that all stores have been performed with respect to other threads or mechanisms that use the storage areas before executing the **stop**. See the the User’s Manual for the implementation for details.

Software must also specify the requested and maximum power-saving level limit fields (i.e RL and MTL fields), and the Transition Rate (TR) field in the PSSCR in order to bound the range of power-saving modes that can be entered. If the value of the RL field is greater than or equal to the value of the MTL field, the power-saving level will not increase from the initial level during power-saving mode.

### Programming Note

If MSR_EE=1 when the **stop** instruction is executed, then the interrupt corresponding to the exception that was expected to cause exit from power-saving mode may occur immediately prior to execution of the **stop** instruction. If this occurs, the result may be a software hang condition since the exception that was expected to cause exit from power-saving mode has already occurred.

The above software hang condition can be prevented by setting MSR_EE=0 prior to executing **stop**.
After the thread has entered power-saving mode with PSSCR EC=1, only the System Reset exception and the exceptions enabled in LPCR PECE and SMFCTRL UDEE will cause exit. If the event that causes exit is a Machine Check exception, then a Machine Check interrupt occurs; otherwise a System Reset interrupt occurs, and the contents of SRR1 indicate the exception that caused exit from power-saving mode. If state loss has occurred in an SMF-enabled system, the interrupt is taken in ultravisor state.

If the hypervisor has set PSSCR SD=0 prior to when the stop instruction is executed, the instruction following the stop may typically be a mfspr in order to read the contents of PSSCR PLS to determine the maximum power-saving level that was entered during power-saving mode.

--- Programming Note ---

If stop was executed when PSSCR EC=0 and MSR CE=0 (in order to avoid the hang condition described in a preceding Programming Note), MSR CE should be set to 1 after power-saving mode is exited in order to take the interrupt corresponding to the exception that caused exit from power-saving mode.

If stop was executed when PSSCR EC=0 and MSREE=0 (in order to avoid the hang condition described in a preceding Programming Note), MSREE should be set to 1 after power-saving mode is exited in order to take the interrupt corresponding to the exception that caused exit from power-saving mode.

--- Programming Note ---

The ultravisor does not initiate power-saving.

If a secure partition attempts to execute stop with parameters that allow state loss, the ultravisor gets control via the Hypervisor Facility Unavailable interrupt. It saves secure state and gives control to the hypervisor’s Hypervisor Facility Unavailable interrupt handler.

Upon exit from a state-losing power-saving mode in an SMF-enabled system, the ultravisor gets control at its Machine Check or System Reset interrupt handler. It restores any ultravisor state that was lost, and then services the Directed Ultravisor Doorbell exception if that caused the wakeup. It then restores the HRMOR and transfers control to the hypervisor’s Machine Check interrupt handler if the ultravisor got control at the ultravisor’s Machine Check interrupt handler, and to the hypervisor’s System Reset interrupt handler otherwise. The hypervisor restores any lost hypervisor state, and then handles the exception (other than Directed Ultravisor Doorbell exception) that caused the wakeup. For this process to work, the ultravisor must have stored a record of its state in some known location prior to transferring control to the hypervisor to execute stop. The hypervisor in turn must have stored its HRMOR value in a location known to the ultravisor. It must also have stored a record of its state in some known location.

The only other function the ultravisor may need to perform for a given power-saving mode transition is to be a proxy accessing hypervisor state in the platform that is mixed with ultravisor state and lacking independent access control.
4.4 Event-Based Branch Facility and Instruction

The Event-Based Branch facility is described in Chapter 6 of Book II, but only at the level required by the application program.

Event-based branches can only occur in problem state and when event-based branches and exceptions have been enabled in the FSCR and HFSCR, and BESCRGE=1. Additionally, the following additional bits must be set to one in order to enable EBB exceptions specific to a given function to occur.

- MMCR0EBE and BESCPME must be set to 1 to enable Performance Monitor event-based exceptions.
- BESCRGE must be set to 1 to enable External event-based exceptions.

If an event-based exception exists (as indicated by BESCRPMEO=1 or BESCREEO=1) when MSRPR=0, the corresponding event-based branch will occur when MSRPR=1, FSCRBB=1, HFSCRBB=1, and BESCRGE=1.

**Programming Note**

Software EBB handlers should ensure that previous exceptions have been cleared (by setting BESCPME0 and/or BESCREEO to 0) before re-enabling event-based branches (by setting BESCRGE to 1 or executing rfebb 1) in order to prevent earlier exceptions from causing additional EBBs.
Chapter 5. Fixed-Point Facility

5.1 Fixed-Point Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Facility that are not covered in Book I.

5.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the \texttt{mfspr} (page 1173) and \texttt{mtspr} (page 1171) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

5.3 Fixed-Point Facility Registers

5.3.1 Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register that contains a value identifying the version and revision level of the implementation. The contents of the PVR can be copied to a GPR by the \texttt{mfspr} instruction. Read access to the PVR is privileged; write access is not provided.

![Figure 9. Processor Version Register](image)

The PVR distinguishes between implementations that differ in attributes that may affect software. It contains two fields.

- **Version**: A 16-bit number that identifies the version of the implementation. Different version numbers indicate major differences between implementations.

- **Revision**: A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between implementations having the same version number, such as clock rate and Engineering Change level.

Version numbers are assigned by the Power ISA process. Revision numbers are assigned by an implementation-defined process.

5.3.2 Processor Identification Register

The Processor Identification Register (PIR) is a 32-bit register that contains a 20-bit PROCID field that can be used to distinguish the thread from other threads in the system. The contents of the PIR can be copied to a GPR by the \texttt{mfspr} instruction. Read access to the PIR is privileged; write access is not provided.

![Figure 10. Processor Identification Register](image)

The means by which the PIR is initialized are implementation-dependent.

The PIR is a hypervisor resource; see Chapter 2.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:43</td>
<td>PROCID</td>
<td>Reserved</td>
</tr>
<tr>
<td>44:63</td>
<td>PROCID</td>
<td>Thread ID</td>
</tr>
</tbody>
</table>

The means by which the PIR is initialized are implementation-dependent.

The PIR is a hypervisor resource; see Chapter 2.
5.3.3 Process Identification Register

The layout of the Process Identification Register (PIDR) is shown in Figure 11 below.

![Figure 11. Process Identification Register](image)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:63</td>
<td>PID</td>
<td>Process Identifier</td>
</tr>
</tbody>
</table>

Access to the PIDR is privileged.

**Programming Note**
Radix tree translation assigns special meaning to PID=0, specifically indicating the operating system's kernel process. When GR=1, PIDR should not be set to zero except when MSRPR=0.

5.3.4 Control Register

The Control Register (CTRL) is a 32-bit register as shown below.

![Figure 12. Control Register](image)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:47</td>
<td>Reserved</td>
</tr>
<tr>
<td>48:55</td>
<td>Thread State (TS)</td>
</tr>
</tbody>
</table>

- Problem State Access
- Privileged Non-hypervisor State Access
- Hypervisor State Access

The RUN bit can be used by the operating system to indicate when the thread is doing useful work.

Write access to the CTRL is privileged. Reads can be performed in privileged or problem state.

5.3.5 Program Priority Register

Privileged programs may set a wider range of program priorities in the PRI field of PPR and PPR32 than may be set by problem state programs (see Chapter 3 of Book II). Problem state programs may only set values in the range of 0b001 to 0b100 unless the Problem State Priority Boost register (see Section 5.3.6) allows the value 0b101. Privileged programs may set values in the range of 0b001 to 0b110. Hypervisor software may also set 0b111. For all priorities except 0b101, if a program attempts to set a value that is not allowed for its privilege level, the PRI field remains unchanged. If a problem state program attempts to set its priority value to 0b101 when this priority value is not allowed for problem state programs, the priority is set to 0b100.

The values and their corresponding meanings are as follows.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11:13</td>
<td>Program Priority (PRI)</td>
</tr>
</tbody>
</table>

- 001 very low
- 010 low
- 011 medium low
- 100 medium
- 101 medium high
- 110 high
- 111 very high

5.3.6 Problem State Priority Boost Register

The Problem State Priority Boost (PSPB) register is a 32-bit register that controls whether problem state pro-
grams have access to program priority medium high. (See Section 3.1 of Book II.)

Figure 13. Problem State Priority Boost Register
A problem state program is able to set the program priority to medium high only when the PSPB of the thread contains a non-zero value.

The maximum value to which the PSPB can be set must be a power of 2 minus 1. Bits that are not required to represent this maximum value must return 0s when read regardless of what was written to them.

When the PSPB is set to a value less than its maximum value but greater than 0, its contents decrease monotonically at the same rate as the SPURR until its contents minus the amount it is to be decreased are 0 or less when a problem state program is executing on the thread at a priority of medium high. When the contents of the PSPB minus the amount it is to be decreased are 0 or less, its contents are replaced by 0.

When the PSPB is set to its maximum value or 0, its contents do not change until it is set to a different value.

Whenever the priority of a thread is medium high and either of the following conditions exist, hardware changes the priority to medium:
- the PSPB counts down to 0, or
- PSPB=0 and the privilege state of the thread is changed to problem state (MSR_{PR}=1).

5.3.7 Relative Priority Register
The Relative Priority Register (RPR) is a 64-bit register that allows the hypervisor to control the relative priorities corresponding to each valid value of PPR_{PRI}.

<table>
<thead>
<tr>
<th>/</th>
<th>RP₁</th>
<th>RP₂</th>
<th>RP₃</th>
<th>RP₄</th>
<th>RP₅</th>
<th>RP₆</th>
<th>RP₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
</tr>
</tbody>
</table>

Figure 14. Relative Priority Register
Each RPₙ field is defined as follows.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:1</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
| 2:7  | Relative priority of priority level n: Specifies the relative priority that corresponds to the priority corresponding to \( \text{PPR}_{PRI}=n \), where a value of 0 indicates the lowest relative priority and a value of 0b11111111 indicates the highest relative priority.

Program Note
The hypervisor must ensure that the values of the RPₙ fields increase monotonically for each n and are of different enough magnitudes to ensure that each priority level provides a meaningful difference in priority.

5.3.8 Software-use SPRs
Software-use SPRs are 64-bit registers provided for use by software.

<table>
<thead>
<tr>
<th></th>
<th>SPRG0</th>
<th>SPRG1</th>
<th>SPRG2</th>
<th>SPRG3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 15. Software-use SPRs
SPRG0, SPRG1, and SPRG2 are privileged registers. SPRG3 is a privileged register except that the contents may be copied to a GPR in Problem state when accessed using the \( \text{mfspr} \) instruction.

Program Note
Neither the contents of the SPRGs, nor accessing them using \( \text{mfspr} \) or \( \text{mfspr} \), has a side effect on the operation of the thread. One or more of the registers is likely to be needed by interrupt handlers that run in privileged non-hypervisor state (e.g., as scratch registers and/or pointers to per thread save areas).

Operating systems must ensure that no sensitive data are left in SPRG3 when a problem state program is dispatched, and operating systems for secure systems must ensure that SPRG3 cannot be used to implement a “covert channel” between problem state programs. These requirements can be satisfied by clearing SPRG3 before passing control to a program that will run in problem state.

HSPRG0 and HSPRG1 are 64-bit registers provided for use by hypervisor programs.

<table>
<thead>
<tr>
<th></th>
<th>HSPRG0</th>
<th>HSPRG1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>
USPRG0 and USPRG1 are 64-bit registers provided for use by ultravisor programs.

<table>
<thead>
<tr>
<th>USPRG0</th>
<th>USPRG1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
</tr>
</tbody>
</table>

Figure 17. SPRs for use by ultravisor programs

Programming Note

Neither the contents of the HSPRGs, nor accessing them using `mtspr` or `mfspr`, has a side effect on the operation of the thread. One or both of the registers is likely to be needed by interrupt handlers that run in hypervisor non-ultravisor state (e.g., as scratch registers and/or pointers to per thread save areas).

Programming Note

Neither the contents of the USPRGs, nor accessing them using `mtspr` or `mfspr`, has a side effect on the operation of the thread. One or both of the registers is likely to be needed by interrupt handlers that run in ultravisor state (e.g., as scratch registers and/or pointers to per thread save areas).
5.4 Fixed-Point Facility Instructions

5.4.1 Fixed-Point Load and Store Caching Inhibited Instructions

The storage accesses caused by the instructions described in this section are performed as though the specified storage location is Caching Inhibited and Guarded. The instructions can be executed only in hypervisor state. Software must ensure that the specified storage location is not in the caches. If the specified storage location is in a cache, the results are undefined.

The Fixed-Point Load and Store Caching Inhibited instructions must be executed only when MSR\textsubscript{DR}=0. The storage location specified by the instructions must not be in storage specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded. If either of these conditions is violated, the result is a Data Storage interrupt.

---

**Programming Note**

The instructions described in this section can be used to permit a control register on an I/O device to be accessed without permitting the corresponding storage location to be copied into the caches.

The Fixed-Point Load and Store Caching Inhibited instructions are fixed-point Storage Access instructions; see Section 3.3.1 of Book I.
**Load Byte and Zero Caching Inhibited Indexed X-form**

\[
\text{lbzcix} \quad \text{RT,RA,RB}
\]

| 31 | RT | RA | RB | 853 | /
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
<td>n1</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if \( RA = 0 \) then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( RT \leftarrow 56 \ || \ \text{MEM}(EA, 1) \)

Let the effective address (EA) be the sum \((RA)0^+ (RB)\). The byte in storage addressed by EA is loaded into \( RT_{56:63} \). \( RT_{0:55} \) are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

**Special Registers Altered:**
None

---

**Load Halfword and Zero Caching Inhibited Indexed X-form**

\[
\text{lhzcix} \quad \text{RT,RA,RB}
\]

| 31 | RT | RA | RB | 821 | /
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
<td>n1</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if \( RA = 0 \) then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( RT \leftarrow 48 \ || \ \text{MEM}(EA, 2) \)

Let the effective address (EA) be the sum \((RA)0^+ (RB)\). The halfword in storage addressed by EA is loaded into \( RT_{48:63} \). \( RT_{0:47} \) are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

**Special Registers Altered:**
None

---

**Load Word and Zero Caching Inhibited Indexed X-form**

\[
\text{lwzcix} \quad \text{RT,RA,RB}
\]

| 31 | RT | RA | RB | 789 | /
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
<td>n1</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if \( RA = 0 \) then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( RT \leftarrow 32 \ || \ \text{MEM}(EA, 4) \)

Let the effective address (EA) be the sum \((RA)0^+ (RB)\). The word in storage addressed by EA is loaded into \( RT_{32:63} \). \( RT_{0:31} \) are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

**Special Registers Altered:**
None

---

**Load Doubleword Caching Inhibited Indexed X-form**

\[
\text{ldcix} \quad \text{RT,RA,RB}
\]

| 31 | RT | RA | RB | 885 | /
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
<td>n1</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if \( RA = 0 \) then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( RT \leftarrow \text{MEM}(EA, 8) \)

Let the effective address (EA) be the sum \((RA)0^+ (RB)\). The doubleword in storage addressed by EA is loaded into RT.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

**Special Registers Altered:**
None
Store Byte Caching Inhibited Indexed X-form

\[
\text{stbcix} \quad RS, RA, RB
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>981</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)
\( EA \leftarrow b + (RB) \)
\( \text{MEM}(EA, 1) \leftarrow (RS)_{56:63} \)

Let the effective address (EA) be the sum \((RA|0)+ (RB)\). \((RS)_{56:63}\) are stored into the byte in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:
None

Store Halfword Caching Inhibited Indexed X-form

\[
\text{sthcix} \quad RS, RA, RB
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>949</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)
\( EA \leftarrow b + (RB) \)
\( \text{MEM}(EA, 2) \leftarrow (RS)_{48:63} \)

Let the effective address (EA) be the sum \((RA|0)+ (RB)\). \((RS)_{48:63}\) are stored into the halfword in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:
None

Store Word Caching Inhibited Indexed X-form

\[
\text{stwcix} \quad RS, RA, RB
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>917</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)
\( EA \leftarrow b + (RB) \)
\( \text{MEM}(EA, 4) \leftarrow (RS)_{32:63} \)

Let the effective address (EA) be the sum \((RA|0)+ (RB)\). \((RS)_{32:63}\) are stored into the word in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:
None

Store Doubleword Caching Inhibited Indexed X-form

\[
\text{stdcix} \quad RS, RA, RB
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>1013</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)
\( EA \leftarrow b + (RB) \)
\( \text{MEM}(EA, 8) \leftarrow (RS) \)

Let the effective address (EA) be the sum \((RA|0)+ (RB)\). \((RS)\) is stored into the doubleword in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:
None
5.4.2 OR Instruction

or Rx,Rx,Rx can be used to set PPRPRI (see Section 5.3.5) as shown in Figure 18. For all priorities except medium high, PPRPRI remains unchanged if the privilege state of the thread executing the instruction is lower than the privilege indicated in the figure. For priority medium high, PPRPRI is set to medium if the thread executing the instruction is in problem state and medium high priority is not allowed for problem state programs. (The encodings available to problem state programs, as well as encodings for additional shared resource hints not shown here, are described in Chapter 3 of Book II.)

<table>
<thead>
<tr>
<th>Rx</th>
<th>PPRPRI</th>
<th>Priority</th>
<th>Privileged</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>001</td>
<td>very low</td>
<td>no</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>low</td>
<td>no</td>
</tr>
<tr>
<td>6</td>
<td>011</td>
<td>medium low</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>medium</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>medium high</td>
<td>no/yes¹</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>high</td>
<td>yes</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>very high</td>
<td>hypv</td>
</tr>
</tbody>
</table>

¹This value is privileged unless the Problem State Priority Boost register allows the priority value 0b101 (See Section 5.3.6.)

Figure 18. Priority levels for or Rx,Rx,Rx

5.4.3 OR Immediate Instruction

ori R31,R31,0 is a no-op instruction that is also execution serializing: that is, executing an ori R31,R31,0 instruction ensures that all instructions preceding the ori R31,R31,0 instruction have completed before the ori R31,R31,0 instruction completes, and that no subsequent instructions are initiated, even out-of-order, until after the ori R31,R31,0 instruction completes.

The ori R31,R31,0 instruction may complete before storage accesses associated with instructions preceding the ori R31,R31,0 instruction have been performed.

Extended Mnemonics:

Additional extended mnemonic for the execution serializing form of Or Immediate:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>exser</td>
<td>ori 31,31,0</td>
</tr>
</tbody>
</table>

Warning: Other forms of ori Rx,Rx,0 that are not described in this section may also have micro-architectural effects on program execution. Use of these forms should be avoided except when software needs the associated micro-architectural effects. If a no-op is needed, the preferred no-op (ori 0,0,0) should be used.

5.4.4 Move To/From System Register Instructions

The Move To Special Purpose Register and Move From Special Purpose Register instructions are described in Book I, but only at the level available to an application programmer. For example, no mention is made there of registers that can be accessed only in privileged state. The descriptions of these instructions given below extend the descriptions given in Book I, but do not list Special Purpose Registers that are implementation-dependent. In the descriptions of these instructions given in below, the “defined” SPR numbers are the SPR numbers shown in the Figure 19 for the instruction and the implementation-specific SPR numbers that are implemented, and similarly for “defined” registers. All other SPR numbers are undefined for the instruction. (Implementation-specific SPR numbers that are not implemented are considered to be undefined.) When an SPR is defined for mtspr and undefined for mfspr, or vice versa, a hyphen appears in the column for the instruction for which the SPR number is undefined.

SPR numbers that are not shown in Figure 19 and are in the ranges shown below are reserved for implementation-specific uses.

<table>
<thead>
<tr>
<th>SPR Numbers for Specific Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>848 - 863</td>
</tr>
<tr>
<td>880 - 895</td>
</tr>
<tr>
<td>976 - 991</td>
</tr>
<tr>
<td>1008 - 1023</td>
</tr>
</tbody>
</table>

Implementation-specific registers must be privileged. SPR numbers for implementation-specific SPRs should be registered in advance with the Power ISA architects.
### Figure 19. SPR encodings (Sheet 1 of 4)

<table>
<thead>
<tr>
<th>SPR( \text{reg} )</th>
<th>SPR( \text{reg} )</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Extended Mnemonics*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00000 0001</td>
<td>XER</td>
<td>no</td>
<td>64</td>
<td>mtßer Rx</td>
</tr>
<tr>
<td>3</td>
<td>00000 0011</td>
<td>DSCR</td>
<td>no</td>
<td>64</td>
<td>mtdscr Rx</td>
</tr>
<tr>
<td>8</td>
<td>00000 0100</td>
<td>LR</td>
<td>no</td>
<td>64</td>
<td>mtlr Rx</td>
</tr>
<tr>
<td>9</td>
<td>00000 0101</td>
<td>CTR</td>
<td>no</td>
<td>64</td>
<td>mtcbr Rx</td>
</tr>
<tr>
<td>13</td>
<td>00000 0110</td>
<td>AMR</td>
<td>no(^4)</td>
<td>64</td>
<td>mtuaamr Rx</td>
</tr>
<tr>
<td>17</td>
<td>00000 1000</td>
<td>DSCR</td>
<td>yes</td>
<td>64</td>
<td>mtdscr Rx</td>
</tr>
<tr>
<td>18</td>
<td>00000 1010</td>
<td>DSISR</td>
<td>yes</td>
<td>32</td>
<td>mtdsir Rx</td>
</tr>
<tr>
<td>19</td>
<td>00000 1011</td>
<td>DAR</td>
<td>yes</td>
<td>64</td>
<td>mtdar Rx</td>
</tr>
<tr>
<td>22</td>
<td>00000 10111</td>
<td>DEC</td>
<td>yes</td>
<td>64</td>
<td>mtdcex Rx</td>
</tr>
<tr>
<td>26</td>
<td>00000 1101</td>
<td>SRR0</td>
<td>yes</td>
<td>64</td>
<td>mtsrr0 Rx</td>
</tr>
<tr>
<td>27</td>
<td>00000 1111</td>
<td>SRR1</td>
<td>yes</td>
<td>64</td>
<td>mtsrr1 Rx</td>
</tr>
<tr>
<td>28</td>
<td>00000 11100</td>
<td>CFAR</td>
<td>yes</td>
<td>64</td>
<td>mtcifar Rx</td>
</tr>
<tr>
<td>29</td>
<td>00000 11111</td>
<td>AMR</td>
<td>yes(^4)</td>
<td>64</td>
<td>mtamr Rx</td>
</tr>
<tr>
<td>48</td>
<td>00001 10000</td>
<td>PDIR</td>
<td>yes</td>
<td>32</td>
<td>mtpdir Rx</td>
</tr>
<tr>
<td>61</td>
<td>00001 11101</td>
<td>IAMR</td>
<td>yes(^4)</td>
<td>64</td>
<td>mtiamr Rx</td>
</tr>
<tr>
<td>152</td>
<td>00100 11000</td>
<td>CTRL</td>
<td>yes</td>
<td>32</td>
<td>mfctrl Rx</td>
</tr>
<tr>
<td>153</td>
<td>00100 11011</td>
<td>FSCR</td>
<td>yes</td>
<td>64</td>
<td>mfscr Rx</td>
</tr>
<tr>
<td>157</td>
<td>00100 11101</td>
<td>UAMOR</td>
<td>yes(^4)</td>
<td>64</td>
<td>mtuamr Rx</td>
</tr>
<tr>
<td>158</td>
<td>00100 11111</td>
<td>na</td>
<td>yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>159</td>
<td>00100 11111</td>
<td>PSPB</td>
<td>yes</td>
<td>32</td>
<td>mtpsbp Rx</td>
</tr>
<tr>
<td>176</td>
<td>00101 10000</td>
<td>DPDES</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mtdpdes Rx</td>
</tr>
<tr>
<td>180</td>
<td>00101 10100</td>
<td>DAWR0</td>
<td>hypv/ult(^{13})</td>
<td>64</td>
<td>mtdawr0 Rx</td>
</tr>
<tr>
<td>181</td>
<td>00101 10101</td>
<td>DAWR1</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mtdawr1 Rx</td>
</tr>
<tr>
<td>186</td>
<td>00101 11010</td>
<td>RPR</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mftrp Rx</td>
</tr>
<tr>
<td>187</td>
<td>00101 11011</td>
<td>CIABR</td>
<td>hypv/ult(^{13})</td>
<td>64</td>
<td>mtciabr Rx</td>
</tr>
<tr>
<td>188</td>
<td>00101 11100</td>
<td>DAWRX0</td>
<td>hypv/ult(^{13})</td>
<td>32</td>
<td>mtdawrx0 Rx</td>
</tr>
<tr>
<td>189</td>
<td>00101 11111</td>
<td>DAWRX1</td>
<td>hypv(^2)</td>
<td>32</td>
<td>mtdawrx1 Rx</td>
</tr>
<tr>
<td>190</td>
<td>00101 11111</td>
<td>HFSCR</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mfhhfscRx</td>
</tr>
<tr>
<td>256</td>
<td>01000 00000</td>
<td>VRSAVE</td>
<td>no</td>
<td>32</td>
<td>mtvrsave Rx</td>
</tr>
<tr>
<td>259</td>
<td>01000 00011</td>
<td>SPRG3</td>
<td>-</td>
<td>64</td>
<td>mfusprg3</td>
</tr>
<tr>
<td>268</td>
<td>01000 01100</td>
<td>TB</td>
<td>-</td>
<td>64</td>
<td>-</td>
</tr>
<tr>
<td>269</td>
<td>01000 01101</td>
<td>TBU</td>
<td>-</td>
<td>32</td>
<td>-</td>
</tr>
<tr>
<td>272-275</td>
<td>01000 100xx</td>
<td>SPRG[n]</td>
<td>yes</td>
<td>64</td>
<td>mtpsgmRx</td>
</tr>
<tr>
<td>284</td>
<td>01000 11100</td>
<td>TBL</td>
<td>hypv(^2)</td>
<td>32</td>
<td>mttbl Rx</td>
</tr>
<tr>
<td>285</td>
<td>01000 11110</td>
<td>TBU</td>
<td>hypv(^2)</td>
<td>32</td>
<td>mttbu Rx</td>
</tr>
<tr>
<td>286</td>
<td>01000 11110</td>
<td>TBU40</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mttbu40 Rx</td>
</tr>
<tr>
<td>287</td>
<td>01000 11111</td>
<td>PVR</td>
<td>yes</td>
<td>32</td>
<td>-</td>
</tr>
<tr>
<td>304</td>
<td>01001 10000</td>
<td>HSPRG0</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mthsprg0 Rx</td>
</tr>
<tr>
<td>305</td>
<td>01001 10001</td>
<td>HSPRG1</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mthsprg1 Rx</td>
</tr>
<tr>
<td>306</td>
<td>01001 10010</td>
<td>HDRSISR</td>
<td>hypv(^2)</td>
<td>32</td>
<td>mthhdrsir Rx</td>
</tr>
<tr>
<td>307</td>
<td>01001 10011</td>
<td>HDAR</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mthdar Rx</td>
</tr>
<tr>
<td>308</td>
<td>01001 11010</td>
<td>SPURR</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mtsfurr Rx</td>
</tr>
<tr>
<td>309</td>
<td>01001 11011</td>
<td>PURR</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mtpurr Rx</td>
</tr>
<tr>
<td>310</td>
<td>01001 11011</td>
<td>HDEC</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mthdec Rx</td>
</tr>
<tr>
<td>313</td>
<td>01001 11011</td>
<td>HRMOR</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mthormor Rx</td>
</tr>
<tr>
<td>314</td>
<td>01001 11010</td>
<td>HSRR0</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mthsrr0 Rx</td>
</tr>
<tr>
<td>315</td>
<td>01001 11011</td>
<td>HSRR1</td>
<td>hypv(^2)</td>
<td>64</td>
<td>mthsrr1 Rx</td>
</tr>
</tbody>
</table>
Figure 19. SPR encodings (Sheet 2 of 4)

<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR$^1$</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Extended Mnemonics$^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td>00001</td>
<td>LPCR</td>
<td>hypv$^2$</td>
<td>64</td>
<td>mtpcr Rx</td>
</tr>
<tr>
<td>319</td>
<td>00001</td>
<td>LPIDR</td>
<td>hypv$^2$</td>
<td>32</td>
<td>mtlipdr Rx</td>
</tr>
<tr>
<td>336</td>
<td>00100</td>
<td>HMER</td>
<td>hypv$^2$</td>
<td>64</td>
<td>mthmer Rx</td>
</tr>
<tr>
<td>337</td>
<td>00100</td>
<td>HMEER</td>
<td>hypv$^2$</td>
<td>64</td>
<td>mthmeer Rx</td>
</tr>
<tr>
<td>338</td>
<td>00100</td>
<td>PCR</td>
<td>hypv$^2$</td>
<td>64</td>
<td>mtcp3r Rx</td>
</tr>
<tr>
<td>339</td>
<td>00100</td>
<td>HEIR</td>
<td>hypv$^2$</td>
<td>32</td>
<td>mehtir Rx</td>
</tr>
<tr>
<td>349</td>
<td>00110</td>
<td>AMOR</td>
<td>hypv$^2$</td>
<td>64</td>
<td>mta3mor Rx</td>
</tr>
<tr>
<td>446</td>
<td>01101</td>
<td>TIR</td>
<td>yes</td>
<td>64</td>
<td>mftir Rx</td>
</tr>
<tr>
<td>464</td>
<td>01110</td>
<td>PTCR</td>
<td>hypv$^{1/2}$</td>
<td>64</td>
<td>mtp3cr Rx</td>
</tr>
<tr>
<td>466</td>
<td>01110</td>
<td>SPRG0</td>
<td>utlv</td>
<td>64</td>
<td>mts3r5p0 Rx</td>
</tr>
<tr>
<td>497</td>
<td>01110</td>
<td>SPRG1</td>
<td>utlv</td>
<td>64</td>
<td>mts3r5p1 Rx</td>
</tr>
<tr>
<td>505</td>
<td>01111</td>
<td>URMOR</td>
<td>utlv</td>
<td>64</td>
<td>mts3r5m0 Rx</td>
</tr>
<tr>
<td>506</td>
<td>01111</td>
<td>USRR0</td>
<td>utlv</td>
<td>64</td>
<td>mts3r5m0 Rx</td>
</tr>
<tr>
<td>507</td>
<td>01111</td>
<td>USRR1</td>
<td>utlv</td>
<td>64</td>
<td>mts3r5m0 Rx</td>
</tr>
<tr>
<td>511</td>
<td>01111</td>
<td>SMFCTRL</td>
<td>utlv</td>
<td>64</td>
<td>mts3r5fct4 R</td>
</tr>
<tr>
<td>736</td>
<td>10111</td>
<td>SIER2</td>
<td>no$^6$</td>
<td>64</td>
<td>mfs3er2 Rx</td>
</tr>
<tr>
<td>737</td>
<td>10111</td>
<td>SIER3</td>
<td>no$^6$</td>
<td>64</td>
<td>mfs3er3 Rx</td>
</tr>
<tr>
<td>738</td>
<td>10111</td>
<td>MMCR3</td>
<td>no$^6$</td>
<td>64</td>
<td>mfs3mcr3 Rx</td>
</tr>
<tr>
<td>752</td>
<td>10111</td>
<td>SIER2</td>
<td>yes</td>
<td>64</td>
<td>mts3ier2 Rx</td>
</tr>
<tr>
<td>753</td>
<td>10111</td>
<td>SIER3</td>
<td>yes</td>
<td>64</td>
<td>mts3ier3 Rx</td>
</tr>
<tr>
<td>754</td>
<td>10111</td>
<td>MMCR3</td>
<td>yes</td>
<td>64</td>
<td>mts3mcr3 Rx</td>
</tr>
<tr>
<td>768</td>
<td>11000</td>
<td>SIER</td>
<td>no$^6$</td>
<td>64</td>
<td>mfs3ier Rx</td>
</tr>
<tr>
<td>769</td>
<td>11000</td>
<td>MMCR2</td>
<td>no$^6$</td>
<td>64</td>
<td>mfs3mar2 Rx</td>
</tr>
<tr>
<td>770</td>
<td>11000</td>
<td>MMCR3</td>
<td>no$^6$</td>
<td>64</td>
<td>mfs3mcr3 Rx</td>
</tr>
<tr>
<td>771</td>
<td>11000</td>
<td>PMC1</td>
<td>no$^6$</td>
<td>32</td>
<td>mfs3pmc1 Rx</td>
</tr>
<tr>
<td>772</td>
<td>11000</td>
<td>PMC2</td>
<td>no$^6$</td>
<td>32</td>
<td>mfs3pmc2 Rx</td>
</tr>
<tr>
<td>773</td>
<td>11000</td>
<td>PMC3</td>
<td>no$^6$</td>
<td>32</td>
<td>mfs3pmc3 Rx</td>
</tr>
<tr>
<td>774</td>
<td>11000</td>
<td>PMC4</td>
<td>no$^6$</td>
<td>32</td>
<td>mfs3pmc4 Rx</td>
</tr>
<tr>
<td>775</td>
<td>11000</td>
<td>PMC5</td>
<td>no$^6$</td>
<td>32</td>
<td>mfs3pmc5 Rx</td>
</tr>
<tr>
<td>776</td>
<td>11000</td>
<td>PMC6</td>
<td>no$^6$</td>
<td>32</td>
<td>mfs3pmc6 Rx</td>
</tr>
<tr>
<td>779</td>
<td>11000</td>
<td>MMCR0</td>
<td>no$^6$</td>
<td>64</td>
<td>mfs3mcr0 Rx</td>
</tr>
<tr>
<td>780</td>
<td>11000</td>
<td>SIAR</td>
<td>no$^6$</td>
<td>64</td>
<td>mfs3iar Rx</td>
</tr>
<tr>
<td>781</td>
<td>11000</td>
<td>SDAR</td>
<td>no$^6$</td>
<td>64</td>
<td>mfs3dar Rx</td>
</tr>
<tr>
<td>782</td>
<td>11000</td>
<td>MMCR1</td>
<td>no$^6$</td>
<td>64</td>
<td>mfs3mar1 Rx</td>
</tr>
<tr>
<td>783</td>
<td>11000</td>
<td>MMCR2</td>
<td>yes</td>
<td>64</td>
<td>mts3mcr2 Rx</td>
</tr>
<tr>
<td>786</td>
<td>11000</td>
<td>MMCR3</td>
<td>yes</td>
<td>64</td>
<td>mts3mcr3 Rx</td>
</tr>
<tr>
<td>787</td>
<td>11000</td>
<td>PMC1</td>
<td>yes</td>
<td>32</td>
<td>mtp3mc1 Rx</td>
</tr>
<tr>
<td>788</td>
<td>11000</td>
<td>PMC2</td>
<td>yes</td>
<td>32</td>
<td>mtp3mc2 Rx</td>
</tr>
<tr>
<td>789</td>
<td>11000</td>
<td>PMC3</td>
<td>yes</td>
<td>32</td>
<td>mtp3mc3 Rx</td>
</tr>
<tr>
<td>790</td>
<td>11000</td>
<td>MMCR0</td>
<td>no$^6$</td>
<td>64</td>
<td>mts3mcr0 Rx</td>
</tr>
</tbody>
</table>

$^1$ SPR = spr$_{5:9}$ spr$_{0:4}$

$^2$ Extended Mnemonics include: mts/mfspr/mtmpr/mtfpr Rx
Figure 19. SPR encodings  (Sheet 3 of 4)

<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR&lt;sup&gt;T&lt;/sup&gt;</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Extended Mnemonics*</th>
</tr>
</thead>
<tbody>
<tr>
<td>790</td>
<td>11000 10110</td>
<td>PMC4</td>
<td>yes</td>
<td>32</td>
<td>mtpmc4 Rx</td>
</tr>
<tr>
<td>791</td>
<td>11000 10111</td>
<td>PMC5</td>
<td>yes</td>
<td>32</td>
<td>mtpmc5 Rx</td>
</tr>
<tr>
<td>792</td>
<td>11000 11000</td>
<td>PMC6</td>
<td>yes</td>
<td>32</td>
<td>mtpmc6 Rx</td>
</tr>
<tr>
<td>795</td>
<td>11000 11011</td>
<td>MMCR0</td>
<td>yes</td>
<td>64</td>
<td>mttmmcr0 Rx</td>
</tr>
<tr>
<td>796</td>
<td>11000 11100</td>
<td>SIAR</td>
<td>yes</td>
<td>64</td>
<td>mtsiar Rx</td>
</tr>
<tr>
<td>797</td>
<td>11000 11101</td>
<td>SDAR</td>
<td>yes</td>
<td>64</td>
<td>mtsdar Rx</td>
</tr>
<tr>
<td>798</td>
<td>11000 11110</td>
<td>MMCR1</td>
<td>yes</td>
<td>64</td>
<td>mttmmcr1 Rx</td>
</tr>
<tr>
<td>800</td>
<td>11001 00000</td>
<td>BESCRS</td>
<td>no</td>
<td>64</td>
<td>mtbescrs Rx</td>
</tr>
<tr>
<td>801</td>
<td>11001 00001</td>
<td>BESCRSU</td>
<td>no</td>
<td>32</td>
<td>mtbescrsu Rx</td>
</tr>
<tr>
<td>802</td>
<td>11001 00010</td>
<td>BESCR</td>
<td>no</td>
<td>64</td>
<td>mtbescrr Rx</td>
</tr>
<tr>
<td>803</td>
<td>11001 00011</td>
<td>BESCRRU</td>
<td>no</td>
<td>32</td>
<td>mtbescrru Rx</td>
</tr>
<tr>
<td>804</td>
<td>11001 00100</td>
<td>EBBHR</td>
<td>no</td>
<td>64</td>
<td>mtebbhr Rx</td>
</tr>
<tr>
<td>805</td>
<td>11001 00101</td>
<td>EBBRR</td>
<td>no</td>
<td>64</td>
<td>mtebbrr Rx</td>
</tr>
<tr>
<td>806</td>
<td>11001 00110</td>
<td>BESCR</td>
<td>no</td>
<td>64</td>
<td>mtbescr Rx</td>
</tr>
<tr>
<td>808</td>
<td>11001 01000</td>
<td>reserved&lt;sup&gt;a&lt;/sup&gt;</td>
<td>no</td>
<td>n/a</td>
<td>-</td>
</tr>
<tr>
<td>809</td>
<td>11001 01001</td>
<td>reserved&lt;sup&gt;a&lt;/sup&gt;</td>
<td>no</td>
<td>n/a</td>
<td>-</td>
</tr>
<tr>
<td>810</td>
<td>11001 01010</td>
<td>reserved&lt;sup&gt;a&lt;/sup&gt;</td>
<td>no</td>
<td>n/a</td>
<td>-</td>
</tr>
<tr>
<td>811</td>
<td>11001 01011</td>
<td>reserved&lt;sup&gt;a&lt;/sup&gt;</td>
<td>no</td>
<td>n/a</td>
<td>-</td>
</tr>
<tr>
<td>815</td>
<td>11001 01110</td>
<td>TAR</td>
<td>no</td>
<td>64</td>
<td>mttar Rx</td>
</tr>
<tr>
<td>816</td>
<td>11001 10000</td>
<td>ASDR</td>
<td>hypv&lt;sup&gt;b&lt;/sup&gt;</td>
<td>64</td>
<td>mtasdr Rx</td>
</tr>
<tr>
<td>823</td>
<td>11010 10111</td>
<td>PSSCR</td>
<td>yes</td>
<td>64</td>
<td>mtppsscr Rx</td>
</tr>
<tr>
<td>848</td>
<td>11010 10000</td>
<td>IC</td>
<td>hypv&lt;sup&gt;b&lt;/sup&gt;</td>
<td>64</td>
<td>mfic Rx</td>
</tr>
<tr>
<td>849</td>
<td>11010 10001</td>
<td>VTB</td>
<td>hypv&lt;sup&gt;b&lt;/sup&gt;</td>
<td>64</td>
<td>mftvb Rx</td>
</tr>
<tr>
<td>855</td>
<td>11010 10111</td>
<td>PSSCR</td>
<td>hypv&lt;sup&gt;b&lt;/sup&gt;</td>
<td>64</td>
<td>mfhpsscr</td>
</tr>
<tr>
<td>896</td>
<td>11100 00000</td>
<td>PPR</td>
<td>no</td>
<td>64</td>
<td>mfppr Rx</td>
</tr>
<tr>
<td>898</td>
<td>11100 00010</td>
<td>PPR32</td>
<td>no</td>
<td>32</td>
<td>mfppr32 Rx</td>
</tr>
<tr>
<td>1023</td>
<td>11111 11111</td>
<td>PIR</td>
<td>-</td>
<td>32</td>
<td>-</td>
</tr>
</tbody>
</table>
**Figure 19. SPR encodings (Sheet 4 of 4)**

<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR ¹</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Extended Mnemonics*</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>spr₅₋₉</td>
<td>spr₀₋₄</td>
<td>mtspr</td>
<td>mfspr</td>
<td>mtspr mfspr</td>
</tr>
</tbody>
</table>

1. This register is not defined for this instruction.
2. Note that the order of the two 5-bit halves of the SPR number is reversed.
3. This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2).
4. This register cannot be directly written. Instead, bits in the register corresponding to 0 bits in (RS) can be cleared using **mtspr** SPR,RS.
5. The value specified in register RS may be masked by the contents of the [U]AMOR before being placed into the AMR; see the **mtspr** instruction description.
6. MMCR₀ᵖMCC and MMCR₀ᵖMCCEXT controls the availability of this SPR, and its contents depend on the privilege state in which it is accessed. See Section 10.4.4 for details.
7. The value specified in Register RS may be masked by the contents of the AMOR before being placed into the IAMR; see the **mtspr** instruction description.
8. The `mftb` instruction is Phased-Out. Assemblers targeting Version 2.03 or later of the architecture should generate an **mfspr** instruction for the `mftb` and `mftbu` extended mnemonics; see the corresponding Assembler Note in the **mftb** instruction description (see Section 5.1 of Book II).
9. No extended mnemonic is provided because previous versions of the architecture defined the obvious extended mnemonic as resolving to the non-privileged SPR number, and because there is no software benefit in using the privileged SPR number, rather than the non-privileged SPR number, for this function.
10. `mfspr` specifying this register is ultravisor privileged when SMFCTRLD=1; otherwise it is hypervisor privileged.

*This figure also defines extended mnemonics for the **mtspr** and **mfspr** instructions, including the Special Purpose Registers (SPRs) defined in Book I and for the **Move From Time Base** instruction defined in Book II.

The **mtspr** and **mfspr** instructions specify an SPR as a numeric operand; extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand. Similar extended mnemonics are provided for the **Move From Time Base** instruction, which specifies the portion of the Time Base as a numeric operand.

**Note:** The `mftb` serves as both a basic and an extended mnemonic. The Assembler will recognize an `mftb` mnemonic with two operands as the basic form, and an `mftb` mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB).
Move To Special Purpose Register

**XFX-form**

```markdown
mtspr SPR,RS

n ← spr[9] || spr[0:4]

switch (n)
  case(13): if MSRHV PR = 0b10 then
              SPR(13) ← (RS)
           else
              if MSRHV PR = 0b00 then
                SPR(13) ← ((RS) & AMOR) |
                           ((SPR(13)) & ¬AMOR)
              else
                SPR(13) ← ((RS) & UAMOR) |
                           ((SPR(13)) & ¬UAMOR)
  case(29,61): if MSRHV PR = 0b10 then
                SPR(n) ← (RS)
           else
                SPR(n) ← ((RS) & AMOR) |
                           ((SPR(n)) & ¬AMOR)
  case (157): if MSRHV PR = 0b10 then
                SPR(157) ← (RS)
           else
                SPR(157) ← (RS) & AMOR
  case (336): SPR(336) ← (SPR(336)) & (RS)
  case (158, 808, 809, 810, 811):
    default: if length(SPR(n)) = 64 then
      SPR(n) ← (RS)
    else
      SPR(n) ← (RS) 32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 19. If the SPR field contains the value 158, the instruction is treated as a privileged no-op. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I. Otherwise, the contents of register RS are placed into the designated Special Purpose Register, except as described in the next five paragraphs. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

When the designated SPR is the Authority Mask Register (AMR), (using SPR 13 or SPR 29), or the designated SPR is the Instruction Authority Mask Register (IAMR), and MSRHV PR=0b00, the contents of bit positions of register RS corresponding to 1 bits in the Authority Mask Override Register (AMOR) are placed into the corresponding bits of the AMR or IAMR, respectively; the other AMR or IAMR bits are not modified.

When the designated SPR is the AMR, using SPR 13, and MSRPR=1, the contents of bit positions of register RS corresponding to 1 bits in the User Authority Mask Override Register (UAMOR) are placed into the corresponding bits of the AMR; the other AMR bits are not modified.

When the designated SPR is the Hypervisor Maintenance Exception Register (HMER), the contents of register RS are ANDed with the contents of the HMER and the result is placed into the HMER.

For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.

spr0=1 if and only if writing the register is privileged. Execution of this instruction specifying an SPR number with spr0=1 when the privilege state of the thread does not permit the access causes one of the following.

- **MSRPR=1**: Privileged Instruction type Program interrupt
- **MSHV PR=0b00 or MSR S HV PR=0b010 and the SPR is always an ultravisor resource (independent of the contents of SMFCTRL)**: Privileged Instruction type Program interrupt
- **MSRHV PR=0b00 and the SPR is a hypervisor resource (see Figure 19) or is PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor privileged for the operation**: Hypervisor Emulation Assistance interrupt

Execution of this instruction specifying an SPR number that is undefined for the implementation causes one of the following.

- if spr0=0:
  - if MSRPR=1: Hypervisor Emulation Assistance interrupt
  - if MSRPR=0: Privileged Instruction type Program interrupt
- if spr0=1:
  - if MSRPR=1: Privileged Instruction type Program interrupt
  - if MSRPR=0: no operation (i.e., the instruction is treated as a no-op) when LPCREVIRT=0 and Hypervisor Emulation Assistance interrupt when LPCREVIRT=1 for all other SPRs

Chapter 5. Fixed-Point Facility  1171
Special Registers Altered:
See Figure 19

---

**Programming Note**

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see Chapter 12, "Synchronization Requirements for Context Alterations" on page 1333.

---

**Programming Note**

Requiring that an attempt to execute an `mtspr` or `mfspr` instruction with SPR=0 or an attempt to execute an `mfspr` instruction with SPR=4, 5, or 6 cause a Hypervisor Emulation Assistance interrupt permits efficient emulation of `mt/lspr` specifying the corresponding SPRs as defined in the POWER Architecture.

Requiring that an attempt to execute an `mtspr` instruction with SPR=4, 5, or 6 cause a Hypervisor Emulation Assistance interrupt, even in privileged state, makes the behavior be the same for both instructions for all four SPR numbers, thereby simplifying the architecture. (SPRs 4, 5, and 6 were not defined for `mtspr` in the POWER Architecture. The corresponding SPRs were privileged for writing, and `mtspr` to those SPRs used the corresponding privileged SPR number.)
Move From Special Purpose Register

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 19. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I. Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

spr0=1 if and only if reading the register is privileged. Execution of this instruction specifying an SPR number with spr0=1 when the privilege state of the thread does not permit the access causes one of the following.

- if MSRPR=0: Hypervisor Emulation Assistance interrupt for SPRs 0, 4, 5, and 6, and no operation (i.e., the instruction is treated as a no-op) when LPCR_EV=0 and Hypervisor Emulation Assistance interrupt when LPCR_EV=1 for all other SPRs
- if spr0=1:
  - if MSRPR=1: Privileged Instruction type Program interrupt
  - if MSRPR=0: no operation (i.e., the instruction is treated as a no-op) when LPCR_EV=0 and Hypervisor Emulation Assistance interrupt when LPCR_EV=1

Special Registers Altered:
None

Note
See the Notes that appear with mt spr.
Move To Machine State Register X-form

Let $L = 0$ then
- $MSR_{48} \leftarrow (RS)_{48} \mid (RS)_{49}$
- $MSR_{58} \leftarrow (RS)_{58} \mid (RS)_{49}$
  - $\neg(MSR_{41} \& MSR_3 \& (\neg (RS)_{49}))$
- $MSR_{59} \leftarrow (RS)_{59} \mid (RS)_{49}$
  - $\neg(MSR_{41} \& MSR_3 \& (\neg (RS)_{49}))$

Let $L = 1$ then
- $MSR_{48, 62} \leftarrow (RS)_{48, 62}$

The MSR is set based on the contents of register RS and of the L field.

$L=0$: The result of ORing bits 48 and 49 of register RS is placed into MSR$_{48}$. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of register RS is complemented and then ANDed with the result of ORing bits 58 and 49 of register RS and placed into MSR$_{58}$. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of register RS is complemented and then ANDed with the result of ORing bits 59 and 49 of register RS and placed into MSR$_{59}$. Bits 32:40, 42:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

$L=1$: Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is privileged.

If $L=0$ this instruction is context synchronizing. If $L=1$ this instruction is execution synchronizing; in addition, the alteration of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:
- MSR

Except in the mtmsr instruction description in this section, references to “mtmsr” in this document imply either $L$ value unless otherwise stated or obvious from context (e.g., a reference to an mtmsr instruction that modifies an MSR bit other than the EE or RI bit implies $L=0$).
Move To Machine State Register
Doubleword  X-form

mtmsrd  RS,L

| 31 | RS | /// | L | /// | 178 | /
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>15</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if L = 0 then

\[
\begin{align*}
\text{MSR}_{48} & \leftarrow (\text{RS})_{48} \lor (\text{RS})_{49} \\
\text{MSR}_{58} & \leftarrow (\text{RS})_{58} \lor (\text{RS})_{49} \\
& \land \neg (\text{MSR}_{41} \land \text{MSR}_3 \land \neg (\text{RS})_{49}) \\
\text{MSR}_{59} & \leftarrow (\text{RS})_{59} \lor (\text{RS})_{49} \\
& \land \neg (\text{MSR}_{41} \land \text{MSR}_3 \land \neg (\text{RS})_{49}) \\
\text{MSR}_{0:2} & \leftarrow 4:40 \lor 42:47 \lor 49:50 \lor 52:57 \lor 60:62 \\
\text{LSB} & \leftarrow (\text{RS})_{0:2} \lor 4:40 \lor 42:47 \lor 49:50 \lor 52:57 \lor 60:62
\end{align*}
\]

else

\[
\text{MSR}_{48} \leftarrow (\text{RS})_{48} \lor 62
\]

The MSR is set based on the contents of register RS and of the L field.

L=0:

The result of ORing bits 48 and 49 of register RS is placed into MSR\(_{48}\). The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of register RS is complemented and then ANDed with the result of ORing bits 58 and 49 of register RS and placed into MSR\(_{58}\). The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of register RS is complemented and then ANDed with the result of ORing bits 59 and 49 of register RS and placed into MSR\(_{59}\). Bits 0:2, 4:40, 42:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is privileged.

If L=0 this instruction is context synchronizing. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:

MSR

Except in the mtmsrd instruction description in this section, references to “mtmsrd” in this document imply either L value unless otherwise stated or obvious from context (e.g., a reference to an mtmsrd instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

---

Programming Note

If this instruction sets MSR\(_{PR}\) to 1, it also sets MSR\(_{EE}\), MSR\(_{IR}\), and MSR\(_{DR}\) to 1. If this instruction results in MSRS\(_{HV}\) being equal to 0b110, it also sets MSR\(_{IR}\) and MSR\(_{DR}\) to 0.

This instruction does not alter MSRS\(_{HV}\), MSRS\(_{S}\), MSRS\(_{ME}\), or MSRS\(_{LE}\).

If the only MSR bits to be altered are MSRS\(_{EE,RI}\), to obtain the best performance L=1 should be used.

---

Programming Note

If MSRS\(_{EE}\)=0 and an External, Decrementer, or Performance Monitor exception is pending, executing an mtmsrd instruction that sets MSRS\(_{EE}\) to 1 will cause the interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 7.9, “Interrupt Priorities” on page 1290). Similarly, if a Hypervisor Decrementer interrupt is pending, execution of the instruction by the hypervisor causes a Hypervisor Decrementer interrupt to occur if HDICE=1.

For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 12.

---

Programming Note

mtmsrd serves as both a basic and an extended mnemonic. The Assembler will recognize an mtmsrd mnemonic with two operands as the basic form, and an mtmsrd mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.
Move From Machine State Register

X-form

\[ \text{mfmsr} \quad \text{RT} \]

\[
\begin{array}{cccccc}
31 & \text{RT} & /// & /// & 83 & \backslash
\end{array}
\]

RT ← MSR

The contents of the MSR are placed into register RT.

This instruction is privileged.

Special Registers Altered:

None
Chapter 6. Storage Control

6.1 Overview
A program references storage using the effective address computed by the hardware when it executes a Load, Store, Branch, or Cache Management instruction, or when it fetches the next sequential instruction. The effective address is translated to a real address according to procedures described in Section 6.7.3, in Section 6.7.7 and in the following sections. The real address is what is presented to the storage subsystem.

For a complete discussion of storage addressing and effective address calculation, see Section 1.10 of Book I.

6.2 Storage Exceptions
A storage exception results when the sequential execution model requires that a storage access be performed but the access is not permitted (e.g., is not permitted by the storage protection mechanism), the access cannot be performed because the effective address cannot be translated to a real address, or the access matches some tracking mechanism criteria (e.g., Data Address Watchpoint).

In certain cases a storage exception may result in the “restart” of (re-execution of at least part of) a Load or Store instruction. See Section 6.7 in this Book.

6.3 Instruction Fetch
Instructions are fetched under control of MSRIR.

MSRIR=0
The effective address of the instruction is interpreted as described in Section 6.7.3.

MSRIR=1
The effective address of the instruction is translated by the Address Translation mechanism described beginning in Section 6.7.7.

6.3.1 Implicit Branch
Explicitly altering certain MSR bits (using mtmsr[if]), or explicitly altering SLB entries, Page Table Entries, or certain System Registers (including the HRMOR, URMOR, and possibly other implementation-dependent registers), may have the side effect of changing the addresses, effective or real, from which the current instruction stream is being fetched. This side effect is called an implicit branch. For example, an mtmsrd instruction that changes the value of MSR_SF may change the effective addresses from which the current instruction stream is being fetched. The MSR bits and System Registers (excluding implementation-dependent registers) for which alteration can cause an implicit branch are indicated as such in Chapter 12. “Synchronization Requirements for Context Alterations” on page 1333. Implicit branches are not supported by the Power ISA. If an implicit branch occurs, the results are boundedly undefined.

6.3.2 Address Wrapping Combined with Changing MSR Bit SF
If the current instruction is at effective address $2^{32} - 4$ and is an mtmsrd instruction that changes the contents of MSR_SF, the effective address of the next sequential instruction is undefined.

Programming Note

If the thread is in 32-bit mode, the current instruction is a word instruction at effective address $2^{32} - 4$ or a prefixed instruction at effective address $2^{32} - 8$, and an interrupt occurs that is defined to set SRR0 or HSRR0 (or LR, for the System Call Vectored interrupt) to the effective address of the next sequential instruction, the contents of SRR0 or HSRR0 (or LR), as appropriate to the interrupt, are undefined.
6.4 Data Access

Data accesses are controlled by MSR\textsubscript{DR}.

\textbf{MSR\textsubscript{DR}}=0

The effective address of the data is interpreted as described in Section 6.7.3.

\textbf{MSR\textsubscript{DR}}=1

The effective address of the data is translated by the Address Translation mechanism described in Section 6.7.7.

6.5 Performing Operations Out-of-Order

An operation is said to be performed “in-order” if, at the time that it is performed, it is known to be required by the sequential execution model. An operation is said to be performed “out-of-order” if, at the time that it is performed, it is not known to be required by the sequential execution model.

Operations are performed out-of-order on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are really needed is contingent on everything that might divert the control flow away from the instruction, such as \textit{Branch, Trap, System Call}, and \textit{Return From Interrupt} instructions, and interrupts, and on everything that might change the context in which the instruction is executed.

Typically, operations are performed out-of-order when resources are available that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or interrupts indicate that the operation would not have been performed in the sequential execution model, any results of the operation are abandoned (except as described below).

In the remainder of this section, including its subsections, “Load instruction” includes the \textit{Cache Management} and other instructions that are stated in the instruction descriptions to be “treated as a Load”, and similarly for “Store instruction”.

A data access that is performed out-of-order may correspond to an arbitrary \textit{Load} or \textit{Store} instruction (e.g., a \textit{Load} or \textit{Store} instruction that is not in the instruction stream being executed). Similarly, an instruction fetch that is performed out-of-order may be for an arbitrary instruction (e.g., the aligned word at an arbitrary location in instruction storage).

Most operations can be performed out-of-order, as long as the machine appears to follow the sequential execution model. Certain out-of-order operations are restricted, as follows.

\textbullet\ Stores are not performed out-of-order (even if the Store instructions that caused them were executed out-of-order).

\textbullet\ Accessing Guarded Storage

The restrictions for this case are given in Section 6.8.1.1.

\textbullet\ Executing instructions subsequent to an \texttt{ori R31,R31,0} instruction

The restrictions for this case are given in Section 5.4.3.

The only permitted side effects of performing an operation out-of-order are the following.

\textbullet\ A Machine Check or Checkstop that could be caused by in-order execution may occur out-of-order.

\textbullet\ Reference and Change bits may be set as described in Section 6.7.12.

\textbullet\ Non-Guarded storage locations that could be fetched into a cache by in-order fetching or execution of an arbitrary instruction may be fetched out-of-order into that cache.

6.6 Invalid Real Address

A storage access (including an access that is performed out-of-order; see Section 6.5) may cause a Machine Check if the accessed storage location contains an uncorrectable error or does not exist.

In the case that the accessed storage location does not exist, the Checkstop state may be entered. See Section 7.5.2 on page 1264.

\begin{center}
\textbf{Programming Note}
\end{center}

In configurations supporting multiple partitions, hypervisor software must ensure that a storage access by a program in one partition will not cause a Checkstop or other system-wide event that could affect the integrity of other partitions (see Chapter 2). For example, such an event could occur if a real address placed in a Page Table Entry does not exist.
6.7 Storage Addressing

Storage Control Overview

- Host real address space size is 2^m bytes, m≤60; see Note 1.
- Guest real address space size is 2^m bytes, m≤60; see Notes 1 and 2.
- Real page size is 2^{12} bytes (4 KB).
- Effective address space size is 2^{64} bytes.

For HPT translation, an effective address is translated to a virtual address via a segment descriptor that was either bolted into the Segment Lookaside Buffer (SLB) by software or found and installed into the SLB via a hardware walk of the Segment Table. After that, the virtual address is translated to a host real address via a hardware walk of the Page Table.
- Virtual address space size is 2^{n} bytes, 65≤n≤78; see Note 3.
- Segment size is 2^{s} bytes, s=28 or 40.
- 2^{n-40} ≤ number of virtual segments ≤ 2^{n-28}; see Note 3.
- Virtual page size is 2^{p} bytes, where 12≤p, and 2^{p} is no larger than either the size of the biggest segment or the real address space; a size of 4 KB, 64 KB, and an implementation-dependent number of other sizes are supported; see Note 3. The Page Table specifies the virtual page size. The SLB specifies the base virtual page size, which is the smallest virtual page size that the segment can contain. The base virtual page size is 2^{b} bytes.
- Segments contain pages of a single size, a mixture of 4 KB and 64 KB pages, or a mixture of page sizes that include implementation-dependent page sizes.

For Radix Tree translation, an effective address is translated to a (guest or host) real address via a hardware walk of the Page Table.
- Virtual page size is 2^{p} bytes, where 12≤p, and 2^{p} is no larger than the size of the real address space; a size of 4 KB, 64 KB, 2MB, and an implementation-dependent number of other sizes are supported; see Note 4. The Page Table specifies the virtual page size. The SLB specifies the base virtual page size, which is the smallest virtual page size that the segment can contain. The base virtual page size is 2^{b} bytes.

Notes:

1. The value of m is implementation-dependent (subject to the maximum given above). When used to address storage or to represent a guest real address, the high-order 60-m bits of the "60-bit" real address must be zeros.

2. The hypervisor may assign a guest real address space size for each partition that uses Radix Tree translation. Accesses to guest real storage outside this range but still mappable by the second level Radix Tree will cause an HiSI or HDSI. Accesses to storage outside the mappable range will have boundedly undefined results.

3. The value of n is implementation-dependent (subject to the range given above). In references to 78-bit virtual addresses elsewhere in this Book, the high-order 78-n bits of the “78-bit” virtual address are assumed to be zeros.

4. The supported values of p for the larger virtual page sizes are implementation-dependent (subject to the limitations given above).

---

Programming Note

Note that without some of the reserved bits in the Radix PTE, the RPN field cannot address the full 60-bit real address space. Similarly without some of the reserved bits in the HPT PTE, the ARPN field cannot address the full 60-bit real address space. Note that without some of the reserved bits in the HPT PTE, the AVA field cannot resolve the full 78-bit virtual address.

6.7.1 32-Bit Mode

The computation of the 64-bit effective address is independent of whether the thread is in 32-bit mode or 64-bit mode. In 32-bit mode (MSRF=0), the high-order 32 bits of the 64-bit effective address are treated as zeros for the purpose of addressing storage. This applies to both data accesses and instruction fetches. It applies independent of whether address translation is enabled or disabled. This truncation of the effective address is the only respect in which storage accesses in 32-bit mode differ from those in 64-bit mode.

---

Programming Note

Treating the high-order 32 bits of the effective address as zeros effectively truncates the 64-bit effective address to a 32-bit effective address such as would have been generated on a 32-bit implementation of the Power ISA. Thus, for example, the ESID in 32-bit mode is the high-order four bits of this truncated effective address; the ESID thus lies in the range 0-15. When address translation is enabled, these four bits would select a Segment Register on a 32-bit implementation of the Power ISA. The SLB entries that translate these 16 ESIDs can be used to emulate these Segment Registers.
6.7.2 Virtualized Partition Memory (VPM) Mode

VPM mode enables the hypervisor to reassign all or part of a partition’s memory transparently so that the reassignment is not visible to the partition. When this is done, the partition’s memory is said to be “virtualized.” This mode is only available within Paravirtualized HPT translation mode. Radix Tree translation mode provides equivalent function by providing two levels of translation with separate Page Tables for the operating system and the hypervisor. (See Section 6.7.7 for a more complete overview of the translation modes.) The VPM field in the LPCR enables VPM mode when address translation is enabled. VPM is always enabled when address translation is disabled.

If the thread is not in hypervisor state, and either address translation is enabled and VPM=1, or address translation is disabled, conditions that would have caused a Data Storage or an Instruction Storage interrupt if the affected memory were not virtualized instead cause a Hypervisor Data Storage or a Hypervisor Instruction Storage interrupt respectively. Because the Hypervisor Data Storage and Hypervisor Instruction Storage interrupts always put the thread in hypervisor state, they permit the hypervisor to handle the condition if appropriate (e.g., to restore the contents of a page that was reassigned), and to reflect it to the operating system’s Data Storage or Instruction Storage interrupt handler otherwise.

When address translation is enabled, VPM mode has no effect on address translation. When address translation is disabled, addressing is controlled as specified in Section 6.7.3.

6.7.3 Ultravisor Real, Hypervisor Real, and Virtual Real Addressing Modes

If a storage access is an instruction fetch performed when instruction address translation is disabled, or if the access is a data access performed when data address translation is disabled, it is said to be performed in “ultravisor real addressing mode” if the thread is in ultravisor state, in “hypervisor real addressing mode” if the thread is in hypervisor non-ultravisor state, and in “virtual real addressing mode” if the thread is in privileged non-hypervisor state. Storage accesses in ultravisor real, hypervisor real, and virtual real addressing modes are performed in a manner that depends on the contents of MSR<sub>HV</sub>, PATE<sub>HR</sub>, PATE<sub>PS</sub>, URMOR (see Chapter 3), HRMOR (see Chapter 2), bit 0 of the effective address (EA<sub>0</sub>), and the state of the Real Mode Storage Control Facility as described below. Bits 1:3 of the effective address are ignored.

MSRs<sub>HV</sub>=0b11
- If EA<sub>0</sub>=0, the Ultravisor Offset Real Mode Address mechanism, described in Section 6.7.3.1, controls the access.
- If EA<sub>0</sub>=1, bits 4:63 of the effective address are used as the real address for the access.

MSRs<sub>HV</sub>=0b01
- If EA<sub>0</sub>=0, the Hypervisor Offset Real Mode Address mechanism, described in Section 6.7.3.1, controls the access.
- If EA<sub>0</sub>=1, bits 4:63 of the effective address are used as the real address for the access.

MSR<sub>HR</sub>=0
- If PATE<sub>HR</sub>=0, the Virtual Real Mode Addressing mechanism, described in Section 6.7.3.3, controls the access.
- If PATE<sub>HR</sub>=1, partition-scoped translation is performed on the effective address. (See Section 6.7.11.3, “Obtaining Host Real Address, Radix on Radix”.)

6.7.3.1 Ultravisor/Hypervisor Offset Real Mode Address

If MSR<sub>HV</sub> = 1 and EA<sub>0</sub> = 0, the access is controlled by the contents of the Ultravisor Real Mode Offset Register or the Hypervisor Real Mode Offset Register, depending on the value of MSR<sub>S</sub>, as follows.

Ultravisor Real Mode Offset Register (URMOR)

When MSR<sub>S</sub>=1, bits 4:63 of the effective address for the access are ORed with the 60-bit offset represented by the contents of the URMOR, and the 60-bit result is used as the real address for the access.

Hypervisor Real Mode Offset Register (HRMOR)

When MSR<sub>S</sub>=0, bits 4:63 of the effective address for the access are ORed with the 60-bit offset represented by the contents of the HRMOR, and the 60-bit result is used as the real address for the access.

For each of the two registers, the supported offset values are all values of the form \(i \times 2^j\), where \(0 \leq i < 2^j\), and \(j\) and \(r\) are implementation-dependent values having the properties that \(12 \leq r \leq 26\) (i.e., the minimum offset granularity is 4 KB and the maximum offset granularity is 64 MB) and \(j + r = m\), where the real address size supported by the implementation is \(m\) bits.
Chapter 6. Storage Control

6.7.3.2 Storage Control Attributes for Accesses in Ultravisor and Hypervisor Real Addressing Modes

Storage accesses in ultravisor and hypervisor real addressing modes are performed as though all of storage had the following storage control attributes, except as modified by the Hypervisor Real Mode Storage Control facility (see Section 6.7.3.2.1). (The storage control attributes are defined in Book II.)

- not Write Through Required
- not Caching Inhibited, for instruction fetches
- not Caching Inhibited, for data accesses except those caused by the Load/Store Caching Inhibited instructions; Caching Inhibited, for data accesses caused by the Load/Store Caching Inhibited instructions
- Memory Coherence Required, for data accesses
- Guarded

Additionally, storage accesses in ultravisor and hypervisor real addressing modes are performed as though all storage was not No-execute.

Programming Note

Because storage accesses in ultravisor and hypervisor real addressing modes do not use the SLB or the Page Table, accesses in these modes bypass all checking and recording of information contained therein (e.g., storage protection checks that use information contained therein are not performed, and reference and change information is not recorded).

6.7.3.2.1 Hypervisor Real Mode Storage Control

The Hypervisor Real Mode Storage Control facility provides a means of specifying portions of real storage that are treated as neither Caching Inhibited nor Guarded in ultravisor and hypervisor real addressing modes (MSR\textsubscript{hyp PR}=0b10, and MSR\textsubscript{RR}=0 or MSR\textsubscript{DR}=0, as appropriate for the type of access). The remaining portions are treated as Caching Inhibited and Guarded in ultravisor and hypervisor real addressing modes.

The means is a hypervisor resource (see Chapter 2), and may also be system-specific.

The facility divides real storage into history blocks, in implementation-specific sizes. The history for instruction fetches is tracked separately from that for data accesses. If there is no instruction fetch history for a block and it is the target of an instruction fetch, the access is performed as though the block is Guarded, but the block is treated as not Guarded for subsequent instruction fetches on a best effort basis, limited by the amount of history that the facility can maintain. If there is no data access history for a block and it is accessed using a Load/Store Caching Inhibited instruction, the access is performed as though the block is Guarded, and the block is treated as Guarded for subsequent accesses on a best effort basis, limited by the amount of history that the facility can maintain. If there is no data access history for a block and it is accessed using any other Load or Store instruction, the access is performed as though the block is Guarded, but the block is treated as not Guarded for subsequent accesses on a best effort basis, limited by the amount of history that the facility can maintain. If the history causes a block to be treated as Guarded, the block is also treated as Caching Inhibited; if the history causes a block to be treated as not Guarded, the block is also treated as not Caching Inhibited.

If the storage location specified by a Load/Store Caching Inhibited instruction is in storage that is specified by the Hypervisor Real Mode Storage Control facility to be treated as not Guarded, a Data Storage interrupt occurs. ("specified by the Hypervisor Real Mode Storage Control facility" means "specified in a history block"). The history can be erased using an slbia instruction; see Section 6.9.3.2.
The facility does not apply to implicit accesses to the Page Table performed during address translation or in recording reference and change information. These accesses are performed as described in Section 6.7.3.4.

6.7.3.3 Virtual Real Mode Addressing Mechanism

If MSR[47]=0, the partition is using Paravirtualized HPT translation (PATEHR=0), and MSR[48]=0 or MSR[49]=0 as appropriate for the type of access, the access is said to be made in virtual real addressing mode and is controlled by the mechanism specified below. The set of storage locations accessible by code is referred to as the Virtualized Real Mode Area (VRMA).

In virtual real addressing mode, address translation, storage protection, and reference and change recording are handled as follows.

- Address translation and storage protection are handled as if address translation were enabled, except that translation of effective addresses to virtual addresses use the SLBE values in Figure 20 instead of the entry in the SLB corresponding to the ESID. In this translation, bits 0:23 of the effective address are ignored (i.e., treated as if they were 0s), bits 24:63-m may be ignored if m < 40, and the Virtual Page Class Key Protection mechanism does not apply.

- Reference and change recording are handled as if address translation were enabled.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESID</td>
<td>360</td>
</tr>
<tr>
<td>V</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0b01 - 1 TB</td>
</tr>
<tr>
<td>VSID</td>
<td>0b00</td>
</tr>
<tr>
<td>Ks</td>
<td>0</td>
</tr>
<tr>
<td>Kp</td>
<td>undefined</td>
</tr>
<tr>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>PATEPS[0]</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>LP</td>
<td>PATEPS[1:2]</td>
</tr>
</tbody>
</table>

The C bit in Figure 20 is set to 0 because the implementation-specific lookaside information associated with the VRMA is expected to be long-lived. See the Programming Note about Class in Section 6.7.8.1.

The 1 TB VSID 0x0_01FF_FFFF should not be used by the operating system for purposes other than mapping the VRMA when address translation is enabled.
Programming Note

Software should specify PTEB = 0b01 for all Page Table Entries that map the VRMA in order to be consistent with the values in Figure 20.

6.7.4 Definitions

translation mode: Refers to either HPT translation or Radix Tree translation. The translation mode is specified by the HR field in the Partition Table Entry corresponding to the contents of the LPIDR.

process-scoped: Refers to translation performed using tables pointed to by Process Table Entries: guest Radix Tree translation, host Radix Tree translation for quadrants 0 and 3 when MSRHV=1, or Segment translation.

partition-scoped: Refers to translation performed using tables found using the first doubleword of Partition Table Entries, either host Radix Tree translation or HPT translation.

fully-qualified address: Refers to the address to be translated, when qualified by the effective LPID and effective PID.

guest real address: Refers to the input to the partition-scoped translation process when using nested Radix Tree translation.

virtual address: Refers to the output of Segment translation and input to HPT translation.

host real address: Refers to the output of the partition-scoped translation process in nested Radix Tree translation or the output of the process-scoped translation in nested Radix Tree translation for quadrants 0 and 3 when MSRHV=1. The simpler “real address” may be used interchangeably.

Page Directory: A table within the Radix Tree translation structure that contains elements (“Page Directory Entries”) that point to other tables, instead of containing just Page Table Entries. The Page Directory that is at the root of the Radix Tree is called the “Root Page Directory.”

effLPID, effPID: This is shorthand for effective LPID and effective PID. In certain circumstances, the value used for the LPID and/or the PID is specified to be zero instead of the actual register contents. “Effective” or “eff” is used to indicate the possibility of such a substitution. This value substitution happens only in Radix Tree translation, and is based on the value of EA0:1 (see Section 6.7.5.1, “Effective Address Space Structure for Radix-using Partitions”). Value substitution does not happen in HPT translation. When a guest uses Radix Tree translation, PID substitution may take place. When a host uses Radix Tree translation, both PID and LPID substitution may take place. When a host uses HPT translation, the only special significance associated with LPIDR=0 is with regard to Segment Table walk when MSRHV=1, as described later.

adjunct: An adjunct is a software entity that resides in a partition along with an operating system and its applications in order to efficiently provide services (e.g., device drivers) for the partition. The adjunct is managed by the hypervisor. It runs in problem state with MSRHVPR=0b011, thereby restricting the resources it can modify (MSRPR=1) and causing its interrupts to go to the hypervisor (MSRHV=0b01). It shares an HPT with the partition it serves. The adjunct’s storage is kept separate from the client partition’s storage using Virtual Page Class Key protection. (The adjunct’s lightness of weight derives from not requiring a full partition context switch (SLB flush, TLB flush, PID/PID change, etc.) when the client partition invokes the services of the adjunct.) Each hardware thread may have its own unique translations for an adjunct. As a result, adjunct segment descriptors cannot exist in the process’s Segment Table and must instead be bolted in the SLB manually. The adjunct construct exists only with an HR=0 hypervisor and only for LPID≠0. The adjunct has its own 64-bit EA space. Entry to an adjunct is only possible from hypervisor state. Prior to dispatching the adjunct, the hypervisor must invalidate SLB entries that map the effective address range that will be used by the adjunct. Similarly, on exit from the adjunct, the hypervisor must invalidate its SLB entries.
6.7.5 Address Ranges Having Defined Uses

The address ranges described below have uses that are defined by the architecture.

- **Fixed interrupt vectors**
  
  Except for the first 256 bytes, which are reserved for software use, the real page beginning at real address 0x0000_0000_0000_0000 is either used for interrupt vectors or reserved for future interrupt vectors.

- **Implementation-specific use**
  
  The two contiguous real pages beginning at real address 0x0000_0000_0000_1000 are reserved for implementation-specific purposes.

- **Offset Real Mode interrupt vectors**
  
  The real pages beginning at the real addresses specified by the URMOR and the HRMOR are used similarly to the page for the fixed interrupt vectors.

- **Relocated interrupt vectors**
  
  Depending on the values of MSR$_{HV}$, MSR$_{IR}$, MSR$_{DR}$ when the interrupt occurs and on the value of LPCR$_{AIL}$, LPCR$_{HAIL}$ as appropriate, the virtual page containing the byte addressed by effective address 0xC000_0000_0000_4000 may be used similarly to the page for the fixed interrupt vectors. (See Section 2.2.)

- **System Call Vectored interrupt vectors**
  
  Depending on the values of MSR$_{HV}$, MSR$_{IR}$, MSR$_{DR}$ when the interrupt occurs and on the value of LPCR$_{AIL}$, LPCR$_{HAIL}$ as appropriate, the virtual page containing the byte addressed by effective address 0xC000_0000_0000_3000 contains the interrupt vectors that are invoked by the System Call Vectored instruction. (See Section 2.2.)

- **Partition Table**
  
  A contiguous sequence of real pages beginning at the real address specified by the PTCR contains the Partition Table.

- **Page Table**
  
  A contiguous sequence of real pages beginning at the real address specified by the first doubleword of the Partition Table Entry when HR=0 contains the Page Table.

6.7.5.1 Effective Address Space Structure for Radix-using Partitions

When Radix Tree translation is in use but translation is disabled (MSR$_{IR}$=0 or MSR$_{DR}$=0, as appropriate for the type of access), MSR$_{HV}$ selects between partition-scoped translation of the real mode guest real address, formed by treating EA$_{0:1}$ as 0b00, and hypervisor or ultravisor real mode (see Section 6.7.3). When Radix Tree translation is in use and translation is enabled, EA$_{0:1}$ together with MSR$_{HV}$ are used to select one of as many as three distinct Radix Trees to which to perform process-scoped translation, as a technique to make system calls and interrupts more efficient by avoiding the need to immediately change the contents of the PIDR and LPIDR. (See Figure 21 for an illustration of the mappings.) Since there's nothing to prevent a process from generating any address in the 64b EA space, the exceptional cases are defined as follows. When a quadrant of the EA space has no associated Radix Tree, access to it results in an Instruction Segment exception or Data Segment exception, as appropriate for the type of access. Similarly, reference to any portion of these quadrants or the real mode guest real address described above that is not mapped by a Radix Tree (versus mapped by an invalid entry) will cause an Instruction or Data Segment exception.

---

**Programming Note**

Note that the quadrant structure is only available to software running in 64b mode with address translation enabled. 32b software will only be able to access storage mapped by its own Radix Tree. When address translation is disabled and HV||PR=0b00, the EA accesses storage mapped into the guest real address space.

---

**Programming Note**

*Warning:* The functionality described in this section, e.g. directing most hypervisor interrupts to the LPIDR=0 translation tables, places great importance on the correctness of the format of and mappings in Partition Table Entry 0 and the tables it anchors. An error in any of these structures could have severe consequences including system checkstops and hangs.

---

**Programming Note**

The intent is that the PIDR and LPIDR contents indicate the process and partition on behalf of which execution is taking place. For example, when a guest process interrupts to the hypervisor, execution to service the interrupt will generally be on behalf of the guest partition. When execution changes to be purely managing hypervisor resources that are not directly tied to any partition, the hypervisor should set LPIDR to 0.

For guest and host applications, guest operating systems, and the hypervisor acting as an operating system (LPIDR=0), quadrant 0 (EA$_{0:1}$=0b00) is mapped by the Radix Tree for the application and quadrant 3 (EA$_{0:1}$=0b11) is mapped by the Radix Tree for the
direct supervisor of the application. Quadrants 1 and 2 have no associated Radix Tree for guest and host applications and guest operating systems, but hold echoes of quadrants 0 and 3 for the hypervisor acting as an operating system.

**Programming Note**

Outboard accelerators may commonly be limited to accessing quadrants 0 and 3 as a matter of platform architecture. In such platforms, references to quadrants 1 and 2 may be regarded as errors.

For the hypervisor acting as a hypervisor (LPIDR ≠ 0), quadrant 3 is as described above. Quadrant 1 (EA0:1=0b01) is mapped by the Radix Tree for the guest application and quadrant 2 (EA0:1=0b10) is mapped by the Radix Tree for the guest operating system, one of which experienced a hypervisor interrupt or performed a system call to the hypervisor. Quadrant 0 has no associated Radix Tree.

When MSRHV=1 and EA0:1=0b00 or 0b11 (and the quadrant is mapped by a Radix Tree), only process-scoped translation is performed. When MSRHV=0 and MSRIR/DR=0, only partition-scoped translation is performed. Otherwise, nested process- and partition-scoped translations are performed.

6.7.6 In-Memory Tables

The In-Memory Tables are used to find the tables that are used in the actual translation process for the partition and process that are executing. They enable hardware, including accelerator hardware separate and distinct from the Power ISA processors in the platform, to perform the translation process largely without software intervention. Description of the In-Memory Table structure follows. Hardware may cache the contents of the In-Memory Tables. Variants of tlbie[1] may be used to manage the caching even though the In-Memory Table contents are not cached in the TLB. When “thread” is used in descriptions of the ordering of accesses and operations (e.g. invalidations) related to translation cache management, it should be understood to include execution streams in accelerators unless otherwise stated or obvious from context.

When an address in the In-Memory Table structure is specified to be a virtual or guest real address, the access to that address is considered to be performed with translation on. For a host using HPT translation, a base page size is specified for each such access to be used in the HPT search. The hypervisor can override the Segment Table Page Size in the Process Table Entry (PRTE, see Figure 24) using LPCRSL. The base page size for the Process Table (PATPRTPS) can be safely altered by the hypervisor since the OS does not have direct access to the Partition Table Entry. All accesses to the In-Memory Tables, the Segment Tables, and the guest Radix Tables that are performed with translation on, including for instruction address translation, are data accesses performed as if MSRPR=0 for the purpose of determining storage protection, although instruction side translation exceptions cause [H]ISI. (A specific example of the implications of this is that tables used to translate instruction fetches may be located in guarded or no-execute storage.)

**Programming Note**

The descriptors in the entries in this section and its subsections contain addresses that are properly aligned so that no shifting is required. For example, the minimum size of the Partition Table is 4KB, so PATB has the thirteenth least significant address bit as its least significant bit. To construct the real address for a 4KB table, 12 zeros are appended on the right, and an appropriate number of address bits are removed from the left to match the real address size (m) supported by the implementation. For an aligned 8K table, bit 51 of the PTCR would be disregarded, and 13 zeros would be appended.

6.7.6.1 Partition Table

The Partition Table Control Register (PTCR) is a 64-bit register that contains the host real address of the base
of the Partition Table and specifies its size. Software must ensure that the contents of the PTCR are the same for all processors in the system prior to enabling translation or transferring control to a partition.

<table>
<thead>
<tr>
<th>///</th>
<th>PATB</th>
<th>///</th>
<th>PATS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>51</td>
<td>58</td>
</tr>
</tbody>
</table>

### Partition Descriptor

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:51</td>
<td>PATB</td>
<td>Partition Table Base</td>
</tr>
<tr>
<td>59:63</td>
<td>PATS</td>
<td>Partition Table Size=$2^{12+PATS}$ PATS≤24</td>
</tr>
</tbody>
</table>

All other fields are reserved.

**Programming Note**

If it becomes necessary to shrink the Partition Table or to change PATB to point to a table that is not identical to the existing one, it is necessary to issue `tlbie` with RIC=2 to invalidate caching of outdated In-Memory Table Entries.

The Partition Table is composed of a pair of double-words per partition. The first doubleword indicates whether the partition uses HPT or Radix Tree translation and whether the partition is secure, and contains the base of the host’s translation table structure in host real memory. The first doubleword also contains the size of the table structure and the size of the Root Page Directory for a hypervisor using Radix Tree translation, or the base page size for the VRMA for Paravirtualized HPT translation. Additional details about the parameters for HPT translation follow.

The HTABORG field contains the high-order 42 bits of the 60-bit real address of the Page Table. The Page Table is thus constrained to lie on a $2^{18}$ byte (256 KB) boundary. At least 11 bits from the hash function (see Figure 31) are used to index into the Page Table. The minimum size Page Table is 256 KB ($2^{11}$ PTEGs of 128 bytes each).

The Page Table can be any size $2^n$ bytes where $18 \leq n \leq 46$. As the table size is increased, more bits are used from the hash to index into the table.

The HTABSIZE field contains an integer giving the number of bits (in addition to the minimum of 11 bits) from the hash that are used in the Page Table index. This number must not exceed 28. HTABSIZE is used to generate a mask of the form 0b00...011...1, which is a string of 28 - HTABSIZE 0-bits followed by a string of HTABSIZE 1-bits. The 1-bits determine which additional bits (beyond the minimum of 11) from the hash are used in the index (see Figure 31).

On implementations that support a real address size of only $m$ bits, $m<60$, bits 0:59-$m$ of the HTABORG field are treated as reserved bits, and software must set them to zeros.

**Example:**

Suppose that the Page Table is 16,384 ($2^{14}$) 128-byte PTEGs, for a total size of $2^{21}$ bytes (2 MB). A 14-bit index is required. Eleven bits are provided from the hash to start with, so 3 additional bits from the hash must be selected. Thus the value in HTABSIZE must be 3. The HPT may begin on any 256KB boundary.
All other fields are reserved.

Figure 23. Partition Table Entry Variants

The second doubleword of the Partition Table Entry contains the base of the partition’s Process Table, which is a guest real address (or effective address when effective LPID=0) for radix hypervisor and virtual address for HPT hypervisor, and the size of the Process Table. The Process Table is assumed to be aligned. Software that uses Radix Tree translation must set the low order PRTS bits of PRTB to 0s. When Segment Tables are provided, the Process Table base address is specified as a VSID with the assumption that the Process Table is located at zero offset in the segment, and also includes the base page size used for the HPT search, with the rest of the implied segment descriptor being B=0b01 (1TB segment), Ks=Kp=0, N=0, C=0, and virtual page class key protection does not apply. The Partition Table Entry variants are illustrated in Figure 23. Note that a configuration with HR=1 for a non-zero LPID and HR=0 for LPID=0 is considered an unsupported MMU configuration because it would attempt to perform HPT translation in quadrants 0 and 3 when MSRHV=1. In addition, LPID=0 with Radix Tree translation is an unsupported MMU configuration when MSRHV=0.

6.7.6.2 Process Table

The Process Table is composed of a quadword Process Table Entry per process in the partition. For partitions that use HPT translation, the Process Table Entry contains a Segment Table descriptor, which is composed of the origin of the Segment Table in virtual address space, the size of the segment and pages that hold the table, the size of the table, and a valid bit that is turned off while changes are made to the entry and Segment Table. The translation of the base address of the Segment Table is completed using an implied segment descriptor with Ks=Kp=0, N=0, C=0, and virtual page class key protection does not apply. For partitions that use Radix Tree translation, the Process Table Entry contains a Radix Tree root descriptor. When running on a host that uses Radix Tree translation, there are two cases. When effLPID=0, the RPDB is a host.
real address. Otherwise, the address is a guest real address and must undergo translation using the hypervisor’s Radix Tree for the partition (i.e. the “partition-scoped” tables, as defined later).

The architecture defines segment translation and two types of page translation. Segment translation is paired with HPT translation. The other supported “pairing” is two level Radix Tree translation. Either of these pairings can be used to translate an effective address into a host real address. The In-Memory Tables described above determine the translation mode used by a partition, as well as the locations of the Page Tables and Segment Tables, and the base page size for the Segment Tables. When MSR_{hv}=1 and/or MSR_{ir}=0 or MSR_{dr}=0 (as appropriate for the type of access), the steps taken for a given mode vary. See Sections 6.7.11.3 and 6.7.11.4 for details.

The pairing of Segment translation and Hashed Page Table (HPT) translation applies Segment translation to an effective address to produce a virtual address as described in Section 6.7.8, and HPT translation to the virtual address to produce a host real address as described in Section 6.7.9. Segment translations can be established by both the guest and the hypervisor, but the HPT translation is always managed by the hypervisor with the guest typically giving direction via system calls to the hypervisor in a paravirtualization relationship. This mode is commonly referred to as Paravirtualized HPT translation. The segment translation is managed on a per-process (“process-scoped”) basis, mapping a smaller effective address space into a large “partition-scoped” virtual address space, where the segment can be used as a shared memory object. There is also the possibility of thread-unique mappings. In the basic version of HPT translation, storage exceptions are directed to the operating system, which in turn issues system calls to the hypervisor. When Virtualized Partition Memory is enabled, storage exceptions are directed to the hypervisor, enabling a higher degree of memory overcommitment as the hypervisor transparently steals pages from the partition. Figure 25 gives an overview of the address translation process.

All other fields are reserved.

Figure 24. Process Table Entry Variants

6.7.7 Address Translation Overview

The effective address (EA) is the address generated by the hardware for an instruction fetch or for a data access. If address translation is enabled, this address is passed to the Address Translation mechanism, which attempts to convert the address to a real address which is then used to access storage. If the effective address cannot be translated, a storage exception (see Section 6.2) occurs.
In Paravirtualized HPT mode, the hypervisor also uses the segment/HPT pairing, and can create a process called an “adjunct”. To do so, it eliminates any potentially conflicting guest segment mappings and creates adjunct mappings prior to dispatching the adjunct.

In the other pairing, Radix Tree translation is used for both the process-scoped and partition-scoped mappings. This mode is sometimes referred to as nested Radix or Radix on Radix translation. Figure 26 gives an overview of the address translation process for Radix on Radix translation. Note that each level of the guest Radix Tree produces a guest real address that must itself undergo partition-scoped translation. See Figure 37 for a detailed illustration of the entire process.

Storage exceptions for process-scoped translation are directed to the operating system, and storage exceptions for partition-scoped translation are directed to the hypervisor. (In this categorization, single level translation is considered process-scoped translation except when VPM is active, in which case it is treated like partition-scoped translation.) As a result, for Radix on Radix translation, the hypervisor can use the partition-scoped mapping to limit the size of the guest real address space, and Virtualized Partition Memory is not necessary to enable a higher degree of memory over-commitment. If in Radix on Radix mode the guest real address is outside the range covered by the partition-scoped Radix Tree, the results are boundedly undefined.

The address specified in ASDR is the guest real address or VSID for which translation has most immediately failed except when the translation fails too early to produce that value. HDAR will generally contain the EA or lower VA bits for which translation has most immediately failed. For example, in the case of a Page Directory being paged out, the ASDR will contain the guest real address of the Page Directory Entry (down to bit 51), rather than the GRA of the datum being accessed. Exceptions may be manifest in unexpected ways. For example, an instruction fetch can fail to set a Change bit in the host PTE mapping the guest PTE. Similarly, the Reference bit update might fail for lack of write authority on the PTE.

Figure 25. Address translation overview

Figure 26. Address translation overview, Radix on Radix

Translation Lookaside Buffer

Conceptually, the Page Table is searched by the address relocation hardware to translate every reference. For performance reasons, the hardware usually
keeps a Translation Lookaside Buffer (TLB) that holds PTEs that have recently been used. The TLB is searched prior to searching the Page Table and, for Radix Tree Translation, prior to searching the Page Walk Cache. As a consequence, when software makes changes to the Page Table it must perform the appropriate TLB invalidate operations to maintain the consistency of the TLB with the Page Table (see Section 6.10). An implementation may associate each of its TLB entries with the partition for which the TLB entry was created, so that the entries can be retained while other partitions are executing.

**Programming Notes**

1. Page Table Entries may or may not be cached in a TLB.
2. It is possible that the hardware implements more than one TLB, such as one for data and one for instructions. In this case the size and shape of the TLBs may differ, as may the values contained therein.
3. Use the `tlbie` instruction to ensure that the TLB no longer contains a mapping for a particular page.

**Page Walk Cache**

For performance reasons, the hardware usually keeps a Page Walk Cache (PWC) that holds Page Directory Entries that represent partial tree traversals (one or more levels) from recent Radix Tree translations. The PWC is searched (perhaps iteratively, depending on the design) with the goal of skipping some of the storage accesses that would otherwise be needed to traverse the Radix Tree. The internal structures of the Radix Trees are considered to be managed separately from the final translations. When software changes this structure, it must perform appropriate invalidations to the PWC to maintain the consistency of the PWC with the Radix Tree (see Section 6.10). An implementation may associate each of its PWC entries with the partition for which the PWC entry was created, so that the entries can be retained while other partitions are executing.

**Programming Notes**

1. Page Directory Entries may or may not be cached in a PWC.
2. It is possible that the hardware implements more than one PWC, such as one for data and one for instructions. In this case the size and shape of the PWCs may differ, as may the values contained therein.
3. Use the `tlbie` instruction to ensure that the PWC no longer contains information describing a particular portion of a Radix Tree.

### 6.7.8 Segment Translation

Conversion of a 64-bit effective address to a virtual address is done by searching the Segment Lookaside Buffer (SLB) as shown in Figure 27. If no matching translation is found in the SLB, LPCR_UPRT=1, and either MSR_{hv}=0 or LPID=0, the Segment Table is searched. For implicit accesses, implicit segment descriptors are provided, as described elsewhere in this chapter.

**Figure 27. Translation of 64-bit effective address to 78 bit virtual address**

### 6.7.8.1 Segment Lookaside Buffer (SLB)

The Segment Lookaside Buffer (SLB) specifies the mapping between Effective Segment IDs (ESIDs) and Virtual Segment IDs (VSIDs). The number of SLB entries is implementation-dependent, except that all implementations provide at least 32 entries.

The first four entries, and when LPCR_UPRT=0 all of the entries, of the SLB are managed by software, using the instructions described in Section 6.9.3.2. See Chapter 12, “Synchronization Requirements for Context Alterations” on page 1333 for the rules that software must follow when updating the SLB.
**SLB Entry**

Each SLB entry (SLBE, sometimes referred to as a “segment descriptor”) maps one ESID to one VSID. Figure 28 shows the layout of an SLB entry.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:35</td>
<td>ESID</td>
<td>Effective Segment ID</td>
</tr>
<tr>
<td>36</td>
<td>V</td>
<td>Entry valid (V=1) or invalid (V=0)</td>
</tr>
<tr>
<td>37:38</td>
<td>B</td>
<td>Segment Size Selector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b00 - 256 MB (s=28)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b01 - 1 TB (s=40)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b10 - reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b11 - reserved</td>
</tr>
<tr>
<td>39:88</td>
<td>VSID</td>
<td>Virtual Segment ID</td>
</tr>
<tr>
<td>89</td>
<td>K_p</td>
<td>Supervisor (privileged) state storage key (see Section 6.7.13.2)</td>
</tr>
<tr>
<td>90</td>
<td>K_s</td>
<td>Problem state storage key (See Section 6.7.13.2.)</td>
</tr>
<tr>
<td>91</td>
<td>N</td>
<td>No-execute segment if N=1</td>
</tr>
<tr>
<td>92</td>
<td>L</td>
<td>Virtual page size selector bit 0</td>
</tr>
<tr>
<td>93</td>
<td>C</td>
<td>Class</td>
</tr>
<tr>
<td>95:96</td>
<td>LP</td>
<td>Virtual page size selector bits 1:2</td>
</tr>
</tbody>
</table>

All other fields are reserved. B_0 (SLBE_{37}) is treated as a reserved field.

**Figure 28. SLB Entry**

Instructions cannot be executed from a No-execute (N=1) segment.

Segments may contain a mixture of page sizes. The L and LP bits specify the base virtual page size for the segment. The SLB\_L\_LP encodings are those shown in Figure 29. The base virtual page size (also referred to as the “base page size”) is the smallest virtual page size that can be used to map a given access, and in most cases is the smallest virtual page size for the segment. (The exception is that multiple base virtual page sizes can occur within the same segment when the base page size specified for a given implicit access (e.g. of one segment table) does not match the base page size specified for another implicit access (e.g. of a different segment table or the process table) or for explicit accesses. References to the base page size for a segment will be understood not to preclude or functionally conflict with this possibility.) The base virtual page size is 2^b bytes. The actual virtual page size (also referred to as the “actual page size” or “virtual page size”) is specified by PTE_{L\_LP}.

<table>
<thead>
<tr>
<th>encoding</th>
<th>base page size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td>4 KB</td>
</tr>
<tr>
<td>0b101</td>
<td>64 KB</td>
</tr>
</tbody>
</table>

Additional values\(^1\) \(2^b\) bytes, where \(b > 12\) and \(b\) may differ among encoding values

\(^1\) The “additional values” are implementation-dependent, as are the corresponding base virtual page sizes. Any values that are not supported by a given implementation are reserved in that implementation.

**Figure 29. Page Size Encodings**

For each SLB entry, software must ensure the following requirements are satisfied.

- \(L||L_P\) contains a value supported by the implementation.
- The base virtual page size selected by the \(L\) and \(L_P\) fields does not exceed the segment size selected by the \(B\) field.
- If \(s=40\), the following bits of the SLB entry contain 0s.
  - \(ESID_{24:35}\)
  - \(VSID_{38:49}\)

The bits in the above two items are ignored by the hardware.

The Class field of the SLBE is used in conjunction with the \textit{slbie}, \textit{slbieg}, and \textit{slbia} instructions (see Section 6.9.3.2). “Class” refers to a grouping of SLB entries and implementation-specific lookaside information so that only entries in a certain group need be invalidated and others might be preserved. The Class value assigned to an implementation-specific lookaside entry derived from an SLB entry must match the Class value of that SLB entry. The Class value assigned to an implementation-specific lookaside entry derived from real mode address “translation,” SLS address translation, or translations required to access the Segment Table Entry Group is 0.

Software must ensure that the SLB contains at most one entry that translates a given effective address, and that if the SLB contains an entry that translates a given effective address, then any previously existing translation of that effective address has been invalidated. An attempt to create an SLB entry that violates this requirement may cause a Machine Check.
6.7.8.2 SLB Search

When the hardware searches the SLB, all entries are tested for a match with the EA. For a match to exist, the following conditions must be satisfied for indicated fields in the SLBE.

- \( V=1 \)
- \( \text{ESID}_{0:63-s} = \text{EA}_{0:63-s} \), where the value of \( s \) is specified by the \( B \) field in the SLBE being tested

If no match is found, the search fails. If one match is found, the search succeeds. If more than one match is found, one of the matching entries is used as if it were the only matching entry, or a Machine Check occurs.

If the SLB search succeeds, the virtual address (VA) is formed from the EA and the matching SLB entry fields as follows.

\[
\text{VA} = \text{VSID}_{0:77-s} \parallel \text{EA}_{64-s:63}
\]

The Virtual Page Number (VPN) is bits 0:77-p of the virtual address. The value of \( p \) is the actual virtual page size specified by the PTE used to translate the virtual address (see Section 6.7.9.1). If \( \text{SLB}_N = 1 \), the \( N \) (No-execute) value used for the storage access is 1.

If the SLB search fails and the state is not such that a Segment Table search will be performed, a segment fault occurs. This is an Instruction Segment exception or a Data Segment exception, depending on whether the effective address is for an instruction fetch or for a data access.

6.7.8.3 Segment Table Description and Search

The Segment Table is an aligned structure composed of 16B segment descriptors organized into 128 byte Segment Table Entry Groups (STEGs). Let \( q = \text{STAB-SIZE}+12, \log_2(\text{size of the Segment Table}) \). The base of the Segment Table in virtual address space is \( \text{STABORG}_{0:77-q} \parallel 0^q \). Software must set the lower order \( q-12 \) bits of \( \text{STABORG} \) to 0s. Primary and secondary hashes are defined for 256MB and 1TB segments, each mapping the ESID to an STEG. The appropriate number (for the size of the Segment Table) of low order ESID bits (their inverse, for the secondary hash) directly select the STEG. The order of STEG specification in the following subsections is the preferred order for a serial search. Implementations may search the STEGs in parallel. If no match is found, a segment fault occurs. If a serial search is done, the search may stop when a match has been found. If more than one match is found, one of the matching entries is used as if it were the only matching entry.

### Programming Note

Class values should be assigned such that Class 0 is used for translations that are expected to be long-lived and Class 1 is used for translations that are expected to be short-lived. This assignment facilitates use of the `slbia` instruction, for which several IH values cause preferential invalidation of Class 1 SLB entries and lookaside information entries.

### Programming Note

It is permissible for software to replace the contents of a valid SLB entry without invalidating the translation specified by that entry provided the specified restrictions are followed. See Chapter 12 Note 10.

### Table

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:35</td>
<td>ESID</td>
<td>Effective Segment ID</td>
</tr>
<tr>
<td>36</td>
<td>V</td>
<td>Entry valid (V=1) or invalid (V=0)</td>
</tr>
<tr>
<td>64:65</td>
<td>B</td>
<td>Segment Size Selector</td>
</tr>
<tr>
<td></td>
<td>0b00</td>
<td>256 MB (s=28)</td>
</tr>
<tr>
<td></td>
<td>0b01</td>
<td>1 TB (s=40)</td>
</tr>
<tr>
<td></td>
<td>0b10</td>
<td>reserved</td>
</tr>
<tr>
<td></td>
<td>0b11</td>
<td>reserved</td>
</tr>
<tr>
<td>66:115</td>
<td>VSID</td>
<td>Virtual Segment ID</td>
</tr>
<tr>
<td>116</td>
<td>( K_s )</td>
<td>Supervisor (privileged) state storage key (see Section 6.7.13.2)</td>
</tr>
<tr>
<td>117</td>
<td>( K_p )</td>
<td>Problem state storage key (See Section 6.7.13.2)</td>
</tr>
<tr>
<td>118</td>
<td>N</td>
<td>No-execute segment if N=1</td>
</tr>
<tr>
<td>119</td>
<td>L</td>
<td>Virtual page size selector bit 0</td>
</tr>
<tr>
<td>120</td>
<td>C</td>
<td>Class</td>
</tr>
<tr>
<td>122:123</td>
<td>LP</td>
<td>Virtual page size selector bits 1:2</td>
</tr>
<tr>
<td>124:127</td>
<td>SW</td>
<td>available for software use</td>
</tr>
</tbody>
</table>

All other fields are reserved.

### Figure 30. Segment Table Entry

6.7.8.3.1 Primary Hash for 256MB Segment

The STEG is located at host VA \( \text{STABORG}_{0:77-q} \parallel \text{EA}_{43-q:35} \parallel 0b00000000 \).

Each of the 8 STEs are searched to find a valid entry (\( V=1, B=0b00 \)) that matches the ESID (\( \text{STE}_{\text{ESID}[0:35]} = \text{EA}_{0:35} \)) of the access being translated.

6.7.8.3.2 Primary Hash for 1TB Segment

The STEG is located at host VA \( \text{STABORG}_{0:77-q} \parallel \text{EA}_{31-q:23} \parallel 0b00000000 \).

Each of the 8 STEs are searched to find a valid entry (\( V=1, B=0b01 \)) that matches the ESID (\( \text{STE}_{\text{ESID}[0:23]} = \text{EA}_{0:23} \)) of the access being translated.

6.7.8.3.3 Secondary Hash for 256MB Segment

The STEG is located at host VA \( \text{STABORG}_{0:77-q} \parallel \sim\text{EA}_{43-q:35} \parallel 0b00000000 \).
Each of the 8 STEs are searched to find a valid entry (V=1, B=0b00) that matches the ESID (STE_ESID[0:35] = EA0:35) of the access being translated.

6.7.8.3.4 Secondary Hash for 1TB Segment

The STEG is located at host VA
STABORG0:77-q || ¬EA31-q:23 || 0b0000000.
Each of the 8 STEs are searched to find a valid entry (V=1, B=0b01) that matches the ESID (STE_ESID[0:23] = EA0:23) of the access being translated.

6.7.9 Hashed Page Table Translation

In Paravirtualized HPT mode, conversion of a 78-bit virtual address to a real address is done by searching the Page Table as shown in Figure 31.
Figure 31. Translation of 78-bit virtual address to 60-bit real address
### 6.7.9.1 Hashed Page Table

The Hashed Page Table (HTAB) is a variable-sized data structure that specifies the mapping between virtual page numbers and real page numbers, where the real page number of a real page is bits 0:47 of the address of the first byte in the real page. The HTAB's size can be any size $2^n$ bytes where $18 \leq n \leq 46$. The HTAB must be located in storage having the storage control attributes that are used for implicit accesses to it (see Section 6.7.3.4). The starting address must be a multiple of $2^{18}$ bytes.

The HTAB contains Page Table Entry Groups (PTEGs). A PTEG contains 8 Page Table Entries (PTEs) of 16 bytes each; each PTEG is thus 128 bytes long. PTEGs are entry points for searches of the Page Table.

See Section 6.10 for the rules that software must follow when updating the Page Table.

---

#### Programming Note

The Page Table must be treated as a hypervisor resource (see Chapter 2), and therefore must be placed in real storage to which only the hypervisor has write access. Moreover, the contents of the Page Table must be such that non-hypervisor software cannot modify storage that contains hypervisor programs or data.

### Page Table Entry

Each Page Table Entry (PTE) maps one VPN to one RPN. Figure 32 shows the layout of a PTE. This layout is independent of the Endian mode of the thread.

<table>
<thead>
<tr>
<th>Dword Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12:56 AVA</td>
<td>Abbreviated Virtual Address</td>
</tr>
<tr>
<td></td>
<td>57:60 SW</td>
<td>Available for software use</td>
</tr>
<tr>
<td></td>
<td>61 L</td>
<td>Virtual page size</td>
</tr>
<tr>
<td></td>
<td>62 H</td>
<td>Hash function identifier</td>
</tr>
<tr>
<td></td>
<td>63 V</td>
<td>Entry valid (V=1) or invalid (V=0)</td>
</tr>
<tr>
<td>1</td>
<td>0 pp</td>
<td>Page Protection bit 0</td>
</tr>
<tr>
<td>2:3</td>
<td>key KEY bits 0:1</td>
<td></td>
</tr>
<tr>
<td>4:5</td>
<td>B</td>
<td>Segment Size</td>
</tr>
<tr>
<td></td>
<td>0b0 - 256 MB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0b01 - 1 TB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0b10 - reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0b11 - reserved</td>
<td></td>
</tr>
</tbody>
</table>

---

#### Programming Note

The AVA field omits the low-order 23 bits of the VA. These bits are not needed in the PTE, because the low-order $b$ of these bits are part of the byte offset into the virtual page and, if $b<23$, the high-order $23-b$ of these bits are always used in selecting the PTEGs to be searched (see Section 6.7.9.2).

A virtual page is mapped to a sequence of $2^{p-12}$ contiguous real pages such that the low-order $p-12$ bits of the real page number of the first real page in the sequence are 0s.

PTEL $L$ specify both a base virtual page size (henceforth referred to as the "base page size") and an actual virtual page size (henceforth referred to as the "actual page size" or "virtual page size"). The actual page size is the size of the virtual page mapped by the PTE. The base page size is the smallest actual page size that a segment can contain for explicit accesses or for a given implicit access, and plays a role in the placement of the PTE in the HPT.

If PTEL $L$=0, the base virtual page size and actual virtual page size are 4KB, and ARPN concatenated with LP (ARPN||LP) contains the page number of the real page that maps the virtual page described by the entry.

---

The H bit in the Page Table Entry should not be set to one unless the secondary Page Table search has been enabled.

---

Because the length of the Abbreviated Virtual Address (AVA) field is only 45 bits, on implementations of this version of the architecture the virtual address size cannot exceed 68 bits ($n \leq 68$). On implementations for which $n<68$, bits 0:67-$n$ of the AVA field must be zeros.

If $b<23$, the AVA field contains bits 10:54 of the VA. Otherwise bits 0:67-$b$ of the AVA field contain bits 10:77-$b$ of the VA, and bits 68-$b$:44 of the AVA field must be zero.

---

Because the length of the Abbreviated Virtual Address (AVA) field is only 45 bits, on implementations of this version of the architecture the virtual address size cannot exceed 68 bits ($n \leq 68$). On implementations for which $n<68$, bits 0:67-$n$ of the AVA field must be zeros.

If $b<23$, the AVA field contains bits 10:54 of the VA. Otherwise bits 0:67-$b$ of the AVA field contain bits 10:77-$b$ of the VA, and bits 68-$b$:44 of the AVA field must be zero.
If \( \text{PTEL}=1 \), the base page size and actual page size are specified by \( \text{PTE}_{LP} \). In this case, the contents of \( \text{PTE}_{LP} \) have the format shown in Figure 33. Bits labelled “r” are bits of the real page number. Bits labelled “z” specify the base page size and actual page size. The values of the “r” bits used to specify each size are implementation-dependent. The values of the “z” bits used to specify each size, along with all possible values of “r” bits in the LP field, must result in LP values distinct from other LP values for other sizes. Actual page sizes 4KB and 64KB are always supported; other actual page sizes are implementation-dependent. If \( \text{PTEL}=1 \), the actual page size must be greater than 4 KB. Which combinations of different base page size and actual page size are supported is implementation-dependent, except that the combination of a base page size of 4 KB with an actual page size of 64 KB is always supported.

The actual page size specified by a given \( \text{PTE}_{LP} \) format is at least \( 2^{12+8-(c)} \), where c is the number of r bits in the format.

### Programming Note

Implementations often have TLBs and implementation-specific lookaside buffers (e.g. ERATs) used to cache translations of recently used storage addresses. Mapping virtual storage to large pages may increase the effectiveness of such lookaside buffers, improving performance, because it is possible for such buffers to translate a larger range of addresses, reducing the frequency that the Page Table must be searched to translate an address.

Instructions cannot be executed from a No-execute (N=1) page.

## Page Table Size

The number of entries in the Page Table directly affects performance because it influences the hit ratio in the Page Table and thus the rate of page faults. If the table is too small, it is possible that not all the virtual pages that actually have real pages assigned can be mapped via the Page Table. This can happen if too many hash collisions occur and there are more than 16 entries for the same primary/secondary pair of PTEGs (when the secondary Page Table search is enabled) or more than 8 entries for the same primary PTEG (when the secondary Page Table search is disabled).

While this situation cannot be guaranteed not to occur for any size Page Table, making the Page Table larger than the minimum size (see Section 6.7.6.1) will reduce the frequency of occurrence of such collisions.

### Programming Note

If large pages are not used, it is recommended that the number of PTEGs in the Page Table be at least half the number of real pages to be accessed. For example, if the amount of real storage to be accessed is \( 2^{31} \) bytes (2 GB), then we have \( 2^{31-12}=2^{19} \) real pages. The minimum recommended Page Table size would be \( 2^{19} \) PTEGs, or \( 2^{25} \) bytes (32 MB).

### 6.7.9.2 Page Table Search

When the hardware searches the Page Table, the accesses are performed as described in Section 6.7.3.4.

An outline of the HTAB search process is shown in Figure 31. Up to two hash functions are used to locate a PTE that may translate the given virtual address.

1. A 39-bit hash value is computed from the VA. The value of s is the value specified in the SLBE that was used to generate the virtual address; the value of b is equal to \( \log_2(\text{base page size specified} \)
in the SLBE that was used to translate the address). **Primary Hash:**

If \( s = 28 \), the hash value is computed by Exclusive ORing \( VA_{11:49} \) with \( (11+b_0||VA_{50:77-b}) \)

If \( s = 40 \), the hash value is computed by Exclusive ORing the following three quantities: \( (VA_{24:37} ||VA_{38:77-b}) \), \( (0||VA_{0:37}) \), and \( (b-10||VA_{38:77-b}) \)

The 60-bit real address of a PTEG is formed by concatenating the following values:
- Bits 0:27 of the 39-bit appropriate primary or secondary hash value ANDed with the mask generated from bits 59:63 of the first doubleword of the Partition Table Entry (HTABSIZE) and then added to the value of bits 4:45 of the first doubleword of the Partition Table Entry (HTABORG).
- Bits 28:38 of the 39-bit hash value.
- Seven 0-bits.

This operation identifies a particular PTEG, called the “primary PTEG”, whose eight PTEs will be tested.

**Secondary Hash:**

If the secondary Page Table search is enabled \( (LPCR_{T}=0) \), perform the secondary hash function as follows; otherwise do not perform step 2 and proceed to step 3 below.

If \( s = 28 \), the hash value is computed by taking the ones complement of the Exclusive OR of \( VA_{11:49} \) with \( (11+b_0||VA_{50:77-b}) \)

If \( s = 40 \), the hash value is computed by taking the ones complement of the following three quantities: \( (VA_{24:37} ||VA_{38:77-b}) \), \( (0||VA_{0:37}) \), and \( (b-10||VA_{38:77-b}) \)

The 60-bit real address of a PTEG is formed by concatenating the following values:
- Bits 0:27 of the 39-bit appropriate primary or secondary hash value ANDed with the mask generated from bits 59:63 of the first doubleword of the Partition Table Entry (HTABSIZE) and then added to the value of bits 4:45 of the first doubleword of the Partition Table Entry (HTABORG).
- Bits 28:38 of the 39-bit hash value.
- Seven 0-bits.

This operation identifies the “secondary PTEG”.

**3. As many as 8 PTEs in the primary PTEG and, if the secondary Page Table search is enabled, 8 PTEs in the secondary PTEG are tested to determine if any translate the given virtual address.**

Let \( q = \min\{54, 77-b\} \). For a match to exist, the following conditions must be satisfied, where \( SLBE \) is the SLBE used to form the virtual address.
- \( PTE_H = 0 \) for the primary PTEG, 1 for the secondary PTEG
- \( PTE_V = 1 \)

### PTEs

- \( PTE_E = SLBE_E \)
- \( PTE_{AV}[0:q-10] = VA_{10:q} \)
- if \( b = 12 \) then
  - \( (PTE_L = 0) \) \( (PTE_{LP} \) specifies the 4KB base page size)  
  - else
  - \( (PTE_L = 1) \) \( (PTE_{LP} \) specifies the base page size specified by \( SLBE_{L||LP} \) )

If no match is found, the search fails. The result is a page fault -- a [Hypervisor] Instruction Storage exception or a [Hypervisor] Data Storage exception, depending on whether the effective address is for an instruction fetch or for a data access. If one match is found, the search succeeds. If more than one match is found, one of the matching entries is used as if it were the only matching entry, or a Machine Check occurs.

If the Page Table search succeeds, the real address \( RA \) is formed by concatenating the following values, where the \( p \) value is the \( \log_2 \) (actual page size specified by \( PTE_{LP} \)).
- three 0 bits
- bits 0:56-p of \( ARPN||LP \) from the matching PTE
- bits 64-p:63 of the effective address (the byte offset)

\[ RA = 0b000 || (ARPN || LP)_{0:56-p} || EA_{64-p:63} \]

A TLB entry may be created as a result of the successful HPT translation. Depending on the specific TLB implementation, the scope of the entry may be the base page size, the virtual page size, or any size in between. In the absence of a TLB, software would be required to create a PTE for each base page sized piece of storage within the virtual page. The number of PTEs actually created to map a virtual page will depend on the scopes supported for TLB entries, the access pattern, and the lifetime of the TLB entries. Hardware generally will not create more than one TLB entry to translate a given virtual address. Multiple matching TLB entries may be created only if the Page Table contains PTEs that map different-sized virtual pages that overlap in the virtual address space. If a TLB search finds multiple matching TLB entries created from such PTEs, one of the matching TLB entries is used as if it were the only matching entry, or a Machine Check occurs. Software should scrupulously avoid creating such mappings.
**Programming Note**

If PTE_L = 0, the actual page size (and base page size) are 4 KB. Otherwise the actual page size and base page size are specified by PTE_LP.

Since hardware searches the Page Table using a value of b equal to \( \log_2 \) (base page size specified in the SLBE that was used to translate the address) regardless of the actual page size, the hardware Page Table search will identify different PTEs for VAs in different 2^b-byte blocks of the virtual page if the actual page size is larger than the base page size. Therefore, there may need to be a valid PTE corresponding to each 2^b-byte block of the virtual page that is referenced. For an actual page size that is larger than \( 2^{23} \) (8 MB), the PTEAVA will differ among some or all of these PTEs. Depending on the Page Table size, some or all of these PTEs may be in the same PTEG. Any such PTEs that are in the same PTEG will differ in the value of PTE_H or PTEAVA or both.

All PTEs for the same virtual page should have the same values in the Page Protection, KEY, ARPN, WIMG, and N fields. A set of values from any one of the PTEs that maps the virtual page may be used for an access in the virtual page since lookaside buffer information may be used to translate the virtual address.

To avoid creating multiple matching PTEs, software should not create PTEs for each of two different virtual pages that overlap in the virtual address space. If the virtual page sizes differ, two virtual pages overlap if the values of virtual address bits 0:77-p for both virtual pages are the same, where \( 2^p \) is the actual virtual page size of the larger page.

**Programming Note**

Because a segment may contain pages of different sizes, the Page Table search uses the segment’s base page size (which is the same for all virtual pages in the segment).

- The value of b used when searching the Page Table to identify the PTEGs to be checked for a match is \( \log_2 \) (segment’s base page size).
- A PTE (in the selected PTEGs) satisfies the Page Table search only if the base page size specified in the PTE is equal to the segment’s base page size.

The matching PTE supplies the actual page size, \( 2^p \); this value of p is used in forming the real address.

A virtual page of \( 2^p \) bytes in a segment with a base page size of \( 2^b \) bytes may be mapped by as many as \( 2^{(p-b)} \) PTEs.

---

**Programming Note**

To obtain the best performance, Page Table Entries should be allocated beginning with the first empty entry in the primary PTEG, or with the first empty entry in the secondary PTEG if the primary PTEG is full and the secondary Page Table search is enabled (LPCR_TC=0).

In Paravirtualized HPT mode, the N (No-execute) value used for the storage access is the result of ORing the N bit from the matching PTE with the N bit from the SLB entry that was used to translate the effective address.

### 6.7.10 Radix Tree Translation

Radix Tree translation uses a nested set of tables to map storage with increasing granularity. Although there is no requirement for an individual table to have uniform content, Page Directories generally contain pointers to other Page Directories or Page Tables (Page Directory Entries, PDEs), while Page Tables are the leaf tables that contain PTEs. Each Page Directory Entry and Page Table Entry in the Radix Tree is 8 bytes long. A Radix Tree root descriptor (RTRD) specifies the size of the address being translated, the size of the root table, and its location. RTRDs appear in variants of the Partition and Process Table Entries. (See Figures 23 and 24.) The Root Page Directory Size (RPDS) is specified as \( \log_2 \) (number of entries in the table). That number of bits is taken from the most significant end of the portion of the address being translated, as an index to choose an element in the Root Page Directory. The entries in the Root Page Directory point to another page of entries, and give its size in the Next Level Size field, PDE_NLS. The most significant NLS bits are taken from the address to choose an entry in that table. The process continues until an entry is found that has its Leaf bit set, indicating it is a Page Table Entry. The base size of the page mapped by the PTE is determined by the number of bits remaining in the address after removing the bits used to select the Page Directory and Page Table Entries. An example with RPDS = 13 and PDE_NLS = 9 in each Page Directory is shown in Figure 34.

The sizes of table supported at each level of the Radix Tree, as well as the ultimate page sizes supported, are implementation specific with the following exceptions. Implementations must support two Radix Tree configurations that map 52 bit effective addresses: each starting with a 64KB root page size followed by 2 levels of 4KB tables, ending with either a 256 byte table or a 4KB table. The former produces a page size of 64KB and the latter a 4KB page size. In both cases, a leaf node in the next to last level of table produces a 2MB page size.
Figure 34. Four level Radix Tree walk translating a 52b EA with NLS=13 in the root PDE and NLS=9 in the other PDEs.

6.7.10.1 Radix Tree Page Directory Entry

...
All other fields are reserved.

Figure 35. Radix Tree Page Directory Entry

6.7.10.2 Radix Tree Page Table Entry

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V</td>
<td>Valid</td>
</tr>
<tr>
<td>1</td>
<td>L</td>
<td>Leaf (entry is a PTE)</td>
</tr>
<tr>
<td>4:55</td>
<td>NLB</td>
<td>Next Level Base</td>
</tr>
<tr>
<td>59:63</td>
<td>NLS</td>
<td>Next Level Size (size of next level of table is $2^{NLS+3}$, $NLS \geq 5$)</td>
</tr>
</tbody>
</table>

All other fields are reserved.

Figure 36. Radix Tree Page Table Entry

6.7.10.3 Nested Translation

When MSR_HV=0 and translation is enabled, each guest real address must undergo partition-scoped translation using the hypervisor’s Radix Tree for the partition. See Figure 37.
When nested translation is being performed, there is the potential for two different sets of protection settings and two different sets of storage attributes. For protection settings, the least permissive values take effect.

For read, write, and execute authority, each is controlled independently based on the least permissive setting of the two translation mechanisms (including all component authority mechanisms within each of them). The Guarded attribute is controlled by the process-scoped PTE. Mismatches of the Caching Inhibited attribute have the following behavior. If the process-scoped PTE specifies I=1 when the parti-
tion-scoped PTE specifies I=0, the result is I=0. The reverse mismatch raises a data storage or instruction storage exception, as appropriate for the access. The results of these rules are shown in Table 5. Together these rules can produce the WIMG=0b0011 state that any individual Att value cannot express.

Unless otherwise stated or obvious from context, references elsewhere in the Books to storage control attributes for nested Radix Tree translations apply to the result of combining the guest and host storage control attributes as specified above. For example, the restrictions on the types of storage that can be accessed by AMOs in Section 4.5 of Book II applies to the results of the combining.

### Table 5: Effective I and G attributes for nested translation

<table>
<thead>
<tr>
<th>process-scoped Att</th>
<th>00</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition-scoped Att</td>
<td>I/G</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>Att mismatch</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>00</td>
<td>10</td>
</tr>
</tbody>
</table>

#### Programming Note

The mismatched Caching Inhibited attribute in the lower left quadrant above is given defined behavior instead of excepting in order to support frame buffer emulation. For frame buffer emulation, the guest believes it is writing to a frame buffer (I=1) in address space that the hypervisor maps to normal memory (I=0).

Reference and Change bit recording is done in both the process-scoped and partition-scoped Page Table Entries. Recording is done as described in Section 6.7.12, “Reference and Change Recording”.

For performance reasons, the result of each walk of a Radix Tree may be cached in a TLB. Logically, the result of each walk is cached separately. For nested translation, the effective to guest real (process-scoped) translation may be cached, as well as the partition-scoped translation for each guest real address produced by the translation process. A minimum of two TLB accesses is required to complete a nested translation: one for the effective to guest real address and one for the guest real to host real address. (An implementation may optimize the process, as long as the optimization can be managed correctly using the `tlbie` instructions that software will use to manage the logical model.)

### 6.7.11 Translation Process

As previously described, in its most complicated form the translation process includes the following steps:
- use of the PTCR to find the required Partition Table Entry
- use of the Partition Table Entry to find the partition-scoped Page Table
- use of the Partition Table Entry and the partition-scoped Page Table to find the required Process Table Entry
- use of the Process Table Entry and partition-scoped Page Table to find the required Segment Table Entry or walk the process-scoped Page Table (i.e. translate the effective address to a virtual or guest real address), and
- use of the partition-scoped Page Table to translate the virtual or guest real address.

Depending on the translation mode and process state, some of these steps may be skipped. The following subsections enumerate the cases and explain the steps in more detail.

#### 6.7.11.1 Fully-Qualified Address

The storage control facilities enable hardware to perform the entire translation process given a “fully-qualified address” and context that makes it a unique input. In addition to its normal use, the term “effective address” is sometimes used as shorthand for the fully-qualified address, and the architecture should be read with this possibility in mind. The following are the components of the fully-qualified address.

- effLPID
- effPID
- EA

The additional context required to perform a translation or match a cached translation may include the following.
- PATEHR (selected using the value in LPIDR, not effLPID)
- MSR_{HV PR IR DR}

The translation mode is selected by the Host Radix bit found in the Partition Table Entry. The Host Radix bit indicates whether the partition is using HPT or Radix Tree translation. Given the overall process, MSR_{HV PR IR DR} determines where and how the process is entered.

#### 6.7.11.2 Finding the Page Tables

[The following description assumes that no legacy mode is active, i.e. LPCRUPRT=1.] The components of the fully-qualified address are used to determine the table(s) used in the translation process. The effective LPID and effective PID are used to find the appropriate Page Table base address(es) using the In-Memory Table structures. Some types of trans-
lation use process-scoped Page Tables, some use partition-scoped Page Tables, and some use both.

Process-scoped table descriptors are found in the Process Tables as follows. The Partition Table Entry (PATE) host real address is calculated by adding the Partition Table Base Address (PATB)\(^{[120]}\) in the PTCR with 16 times the effective LPID. The second doubleword of the entry contains the base address of the Process Table for the partition. The Process Table is assumed to be aligned in effective \((HR=1, effLPID=0)\), virtual, or guest real address space. The Process Table Entry (PRTE) host real address is calculated by ORing the Process Table Base Address (PRTB)\(^{[400]}\) for an HPT host and PRTB\(^{[120]}\) for a radix host) in the PATE with 16 times the effective PID and then performing partition-scoped translation. (If the table is not aligned or is not large enough to support the PID value, an unreported error will most likely result.) The Process Table Entry at that location contains a process-scoped table base address, which is a guest real address for a radix guest \((HV=0)\), a host real address for a radix host \((HV=1)\), or a virtual address (all cases using HPT translation). The virtual or guest real address must be translated via the appropriate partition-scoped table.

### Programming Note

The guest real or virtual address of the Process Table, for a radix or HPT guest, respectively, may be set via an hcall. The radix guest may choose to map the Process Table into its own effective address space. These matters are not visible to the architecture.

### Programming Note

Note that the sole purpose of partition-scoped Page Table descriptor when LPID=0 for a radix host is to translate the effective addresses of the Process Table Entries for LPID=0. (If the Process Table Base address for LPID=0 was a real address, the Process Table would have to be in contiguous real storage.) This descriptor will commonly be the same as the descriptor found in the LPID=0, PID=0 Process Table Entry, both pointing to the hypervisor’s own page trable, but it may be set up to point to a table used solely to translate the addresses of Process Table Entries.

Partition-scoped Page Table descriptors are found in the Partition Table as follows. The Partition Table Base Address is found in the PTCR. The effective LPID (times 16 bytes per partition) is used to index off the Partition Table Base Address to find the appropriate Partition Table Entry. The first doubleword of the entry contains the base address of the Page Table.

### 6.7.11.3 Obtaining Host Real Address, Radix on Radix

The following cases exist.
- Guest access to quadrant 0 with translation on: process-scoped translation is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation with effective LPID=LPIDR.
- Guest access to quadrant 3 with translation on: process-scoped translation is performed on LPIDR||0||EA, with the result subject to partition-scoped translation with effective LPID=LPIDR.
- Hypervisor access to quadrant 1 with translation on: process-scoped translation is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation with effective LPID=LPIDR.
- Hypervisor access to quadrant 2 with translation on: process-scoped translation is performed on LPIDR||0||EA, with the result subject to partition-scoped translation with effective LPID=LPIDR.
- Guest OS access with translation off: partition-scoped translation is performed with effective LPID = LPIDR.
- Hypervisor or host application access to quadrant 0 with translation on: process-scoped translation is performed on 0||PIDR||EA.
- Hypervisor or host application access to quadrant 3 with translation on: process-scoped translation is performed with 0||0||EA.
- Hypervisor or ultravisor real mode access: subject to EA 0 and either HRMOR or URMOR, as described in Section 6.7.3.1.
6.7.11.4 Obtaining Host Real Address, HPT

There are two scenarios for Paravirtualized HPT translation. The first is the legacy scenario with a native HPT hypervisor. The second scenario is for a Radix Tree translation hypervisor providing a Paravirtualized HPT environment for the guest. In this latter scenario, the LPID=0 Partition Table Entry will have HR=1. For both scenarios the LPID value is always taken from LPIDR and the PID value is always taken from PIDR, even when MSR_HV=1. In the latter scenario, the hypervisor will explicitly set LPIDR=0 when it wants to use its Radix Tree(s).

When using Paravirtualized HPT translation, the process-scoped Page Tables are replaced by Segment Tables, and the description in Section 6.7.11.2, “Finding the Page Tables” can be read with that substitution in mind. The process-scoped translation is the effective-to-virtual translation described in Section 6.7.8. In-Memory Table walks are processed via the LPID=LPIDR partition-scoped HPT.

As with the previous enumerations, this is done from a hardware point of view. As a result, it does not differentiate the software cases for which Segment translation should only be satisfied by bolted translations.

The following cases exist.

- Guest access with translation on: process-scoped translation is performed on LPIDR||PIDR||EA with the result subject to partition-scoped translation using parameters from the matching segment descriptor.
- Hypervisor or adjunct access with translation on and LPID=0: process-scoped translation, limited to an SLB search with no Segment Table walk, is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation using parameters from the matching segment descriptor.
- Hypervisor or adjunct access with translation on and LPID=0: process-scoped translation (with Segment Table walk) is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation using parameters from the matching segment descriptor.
- Guest OS access with translation off: subject to VPM, as described in Section 6.7.3.3.
- Hypervisor or ultravisor real mode access: subject to EA0 and either HRMOR or URMOR, as described in Section 6.7.3.1.

Figure 38. Radix on Radix translation, general case

Figure 39. Paravirtualized HPT translation

6.7.12 Reference and Change Recording

When operating in Paravirtualized HPT mode, Reference (R) and Change (C) bits are updated in any one of what could be multiple (because of the multiple base size PTEs mapping a virtual page) Page Table Entries that map the virtual page that is being accessed. When operating in Radix on Radix mode, Reference (R) and
Change (C) bits may be updated in multiple Page Table Entries that are accessed as part of the translation process. (For example, each access to a guest’s Page Directory or Page Table Entry potentially sets a Reference bit in the partition-scoped table mapping it.) If the storage operand of a Load or Store instruction crosses a virtual page boundary, the accesses to the components of the operand in each page are treated as separate and independent accesses to each of the pages for the purpose of setting the Reference and Change bits.

For Radix Tree translation, hardware attempts to set the Reference and Change bits atomically, as though the PTE was read to perform the translation using a Load And Reserve instruction, and conditional on the translation being valid and correct (and on the existence of the reservation), the appropriate bit(s) are set as though with a Store Conditional instruction. ("as though" indicates that the reservation(s) held for this purpose are distinct from one another and from the reservation established by a Load And Reserve instruction.) If hardware is unable to set the bit(s) atomically, a [Hypervisor] Data Storage or [Hypervisor] Instruction Storage interrupt will be caused. For HPT translation, hardware sets the Reference and Change bits as though the PTE was read to perform the update using a (simple) Load instruction and the appropriate bit(s) are set as though with a (simple) Store instruction.

For both kinds of translation, setting the bits need not be atomic with respect to performing the access that causes the bits to be updated. The Reference bit must contain 1 in order to load from the corresponding page. The Change bit must contain 1 in order to store to the corresponding page.

**Programming Note**

The interrupt indicates to software that it must set the appropriate bit(s) itself. Note that an instruction fetch can cause a Change bit to be set, for example in the host Page Table Entry that maps the guest Page Table Entry if the instruction fetch causes the Reference bit to be set in the guest Page Table Entry.

**Programming Note**

In nested Radix Tree translation, as many as three Change bits may be set: in the process-scoped and partition-scoped PTEs for the access itself, and in the partition-scoped PTE that maps the process-scoped PTE. Similarly, a large number of Reference bits may be set, including for each partition-scoped PTE that maps a process-scoped PDE or PTE.

Reference and Change bits are set by the hardware as described below. An attempt to access storage may cause one or more of the bits to be set (as described below) even if the access is not performed. The bits are updated in the Page Table Entry if the new value would otherwise be different from the old value for the virtual page, as determined by examining either the Page Table Entry or any lookaside information for the virtual page (e.g., TLB) maintained by the hardware.

**Reference Bit**

The Reference bit is set to 1 if the corresponding access (load, store, implicit access, or instruction fetch) is required by the sequential execution model and is performed. Otherwise the Reference bit may be set to 1 if the corresponding access is attempted, either in-order or out-of-order, even if the attempt causes an exception, except that the Reference bit is not set to 1 for the access caused by an indexed Move Assist instruction for which the XER specifies a length of zero.

**Change Bit**

The Change bit is set to 1 if a Store instruction is executed and the store is performed or if an implicit update is performed. Otherwise in general
the Change bit may be set to 1 if a Store instruction is executed and the store is permitted by the storage protection mechanism and, if the Store instruction is executed out-of-order, the instruction would be required by the sequential execution model in the absence of the kinds of interrupts:

- system-caused interrupts (see Section 7.4 on page 1255)
- Floating-Point Enabled Exception type Program interrupts when the thread is in an Imprecise mode.

The only exceptions to the preceding statement are that the Change bit is not set to 1 if the instruction is a Store String Indexed instruction for which the XER specifies a length of zero, if the instruction is a Load Atomic or Store Atomic instruction with an invalid function code, or if the instruction is a Store Caching Inhibited instruction executed when MSRDR=1.

When the hardware updates the Reference and Change bits in a Page Table Entry, the accesses are performed as described in Section 6.7.3.4, “Storage Control Attributes for Implicit Storage Accesses” on page 1183. These Reference and Change bit updates are not necessarily immediately visible to software. Executing a sync instruction ensures that all Reference and Change bit updates associated with address translations that were performed, by the thread executing the sync instruction, before the sync instruction is executed will be performed with respect to that thread before the sync instruction’s memory barrier is created. There are additional requirements for synchronizing Reference and Change bit updates in multi-threaded systems; see Section 6.10, “Translation Table Update Synchronization Requirements” on page 1241.

---

**Programming Note**

A virtual page in a segment with a smaller base page size may be mapped by multiple PTEs. For each access of a virtual page, hardware may search the Page Table to update the R and C bits. If lookaside buffer information for the virtual page already indicates that all such bits to be set have already been set in a PTE that maps the virtual page, hardware need not make an update. Consider the following sequence of events:

1. A virtual page is mapped by 2 PTEs A and B and the R and C bits in both PTEs are 0.
2. A Load instruction accesses the virtual page and the R bit is updated in PTE A.
3. A Load instruction accesses the virtual page and the R bit is updated in PTE B.
4. A Store instruction accesses the virtual page and the C bit is updated in PTE B.
5. The virtual page is paged out. Software must examine both PTE A and B to get the state of the R and C bits for the virtual page.

Furthermore, if in event 2, PTE A was not found, a Data Storage interrupt or Hypervisor Data Storage interrupt may occur. Subsequently, if in event 3 or 4, PTE B was not found, a Data Storage interrupt or Hypervisor Data Storage interrupt may occur.

---

Even though the execution of a Store instruction causes the Change bit to be set to 1, the store might not be performed or might be only partially performed in cases such as the following.

- A Store Conditional instruction (stwcx. or stdcx.) or a Load Atomic or Store Atomic instruction (e.g. Fetch and Increment Bounded, Store Twin) is executed, but no store is performed.
- The Store instruction causes a Data Storage exception (all cases except Load Atomic or Store Atomic with an invalid function code, Store Caching Inhibited executed when MSRDR=1, EAO, or storage protection violation, which do not store and are not permitted to set the Change bit).
- The Store instruction causes an Alignment exception.
- The Page Table Entry that translates the virtual address of the storage operand is altered such that the new contents of the Page Table Entry preclude performing the store (e.g., the PTE is made invalid, or the PP bits are changed).

For example, when executing a Store instruction, the thread may search the Page Table for the purpose of setting the Change bit and then re-execute the instruction. When reexecuting the instruction, the thread may search the Page Table a second time. If the Page Table Entry has meanwhile been altered, by a program executing on another thread, the second search may obtain the new contents, which may preclude the store.

- A system-caused interrupt occurs before the store has been performed.
If software refers to a Page Table Entry when MSRDR=1 or MSR HV=0, the Reference and Change bits in the associated Page Table Entry are set as for ordinary loads and stores. See Section 6.10 for the rules software must follow when updating Reference and Change bits.

Figure 40 on page 1207 summarizes the rules for setting the Reference and Change bits. The table applies to each atomic storage reference. It should be read from the top down; the first line matching a given situation applies. For example, if stwcx fails due to both a storage protection violation and the lack of a reservation, the Change bit is not altered. The figure applies to PTE(s) that map instructions or storage operands of instructions. When Radix Tree translation is in use, Reference and Change bits are set in other, partition-scoped, PTEs as described earlier in this section.

In the figure, the “Load-type” instructions are the Load instructions described in Books I, II, and III, and the Cache Management instructions that are treated as Loads. The “Store-type” instructions are the Store instructions described in Books I, II, and III, and the Cache Management instructions that are treated as Stores. The Load Atomic and Store Atomic instructions are considered to be both loads and stores, and as a result could match “Load-type” and “Store-type” entries in the table. As a result, “Store-type” entries precede “Load-type” entries in the table so that AMOs match “Store-type” entries. The “ordinary” Load and Store instructions are those described in Books I, II, and III. “set” means “set to 1”.

Because the sync instruction is execution synchronizing, the set of Reference and Change bit updates that are performed with respect to the thread executing the sync instruction before the memory barrier is created includes all Reference and Change bit updates associated with instructions preceding the sync instruction.

Programming Note

Because the sync instruction is execution synchronizing, the set of Reference and Change bit updates that are performed with respect to the thread executing the sync instruction before the memory barrier is created includes all Reference and Change bit updates associated with instructions preceding the sync instruction.

<table>
<thead>
<tr>
<th>Status of Access</th>
<th>R</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed Move Assist insn w 0 len in XER</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Load or Store Atomic instruction with invalid function code, Load or Store Caching Inhibited executed when MSRDR=1</td>
<td>Acc(^1)</td>
<td>No</td>
</tr>
<tr>
<td>Storage protection violation</td>
<td>Acc(^1)</td>
<td>No</td>
</tr>
<tr>
<td>Out-of-order Store-type inst'n, excluding dcbtst</td>
<td>Acc</td>
<td>Acc(^1) 2</td>
</tr>
<tr>
<td>Would be required by the sequential execution model in the absence of system-caused or imprecise interrupts(^3)</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>All other cases</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>Out-of-order I-fetch or Load-type Inst'n (including dcbtst)</td>
<td>Acc</td>
<td>Acc(^2)</td>
</tr>
<tr>
<td>In-order Load-type or Store-type insn, access not performed(^4)</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>Store-type insn</td>
<td>Acc</td>
<td>Acc(^2)</td>
</tr>
<tr>
<td>Load-type insn</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>Other in-order access</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Other ordinary Store, dcbz</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>icbi, icbt, dcbt, dcbtst, dcbst, dcbf</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>l-fetch or ordinary Load</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

*R means that it is acceptable to set the bit.

\(^1\) It is preferable not to set the bit.

\(^2\) If C is set, R is also set unless it is already set.

\(^3\) For Floating-Point Enabled Exception type Program interrupts, “imprecise” refers to the exception mode controlled by MSRFE0 FE1.

\(^4\) This case does not apply to the Touch instructions, because they do not cause a storage access.

Figure 40. Setting the Reference and Change bits
6.7.13 Storage Protection

The storage protection mechanism provides a means for selectively granting instruction fetch access, granting read access, granting write access, and prohibiting access to areas of storage based on a number of control criteria.

The operation of the storage protection mechanism depends on the value of one or more of the following.

- MSR bits HV, S, IR, DR, PR
- the key bits and N bit in the associated SLB entry
- the page protection bits, key bits, N bit, and G attribute in the associated PTE
- the AMR, IAMR, AMOR, and UAMOR
- the Secure Memory property

The storage protection mechanism consists of the Virtual Page Class Key Protection mechanism described in Section 6.7.13.1, the Basic Storage Protection mechanism described in Section 6.7.13.2 and Section 6.7.13.3, the Radix Tree Translation Storage Protection mechanism described in Section 6.7.13.4, and the Secure Memory Protection mechanism described in Section 6.7.13.5.

In order for a storage access to be permitted, it must be permitted by all of the mechanisms that apply to it. If SMFCTRLE=1, each storage access is subject to Secure Memory Protection independent of the translation mode of the access. In addition, each access is subject to other protection mechanisms depending on its translation mode, as listed below.

- **MSR might**=1 and address translation is disabled: Basic Storage Protection mechanism
- **HR**=0
  - access to instruction or data when address translation is enabled: Virtual Page Class Key Protection mechanism and Basic Storage Protection mechanism
  - all other cases (access to Process Table Entry or Segment Table Entry when address translation is enabled; access to instruction or data when MSR might=0 and address translation is disabled): Basic Storage Protection mechanism

**Programming Note**

Because the assumed K s and K p values are either 0 or irrelevant, these accesses are always permitted by the Basic Storage Protection mechanism.

- **HR**=1

- access to instruction or data when address translation is enabled and effLPID≠0: Radix Tree Translation Storage Protection mechanisms of both the process-scoped and partition-scoped PTEs, except that the Guarded attribute (which affects storage protection for instruction fetches) is determined solely by the process-scoped PTE

- access to instruction or data when address translation is enabled and effLPID=0: Radix Tree Translation Storage Protection mechanism of the process-scoped PTE

- all other cases (access to Process Table Entry when address translation is enabled; access to process-scoped PDE or process-scoped PTE when address translation is enabled and effLPID=0; access to instruction or data when MSR might=0 and address translation is disabled): Radix Tree Translation Storage Protection mechanism of the partition-scoped PTE

If an access associated with an instruction fetch is not permitted, an Instruction Storage exception or a Hypervisor Instruction Storage exception is generated. If an access associated with a data access is not permitted, a Data Storage exception or a Hypervisor Data Storage exception is generated.

A **protection domain** is a maximal range of effective addresses, virtual addresses, or guest real addresses for which variables related to storage protection can be independently specified (including by default, as in virtual real, hypervisor real, and ultravisor real addressing modes), or a maximal range of addresses, effective, virtual, or guest real, for which variables related to storage protection cannot be specified. Examples include: a segment, a virtual page (including for the Virtualized Real Mode Area), the Virtualized Real Mode Area, the effective address range 0:260-1 in hypervisor and ultravisor real addressing modes, and a maximal range of effective, virtual, or guest real addresses that cannot be mapped to real addresses. A **protection boundary** is a boundary between protection domains.

6.7.13.1 Virtual Page Class Key Protection

The Virtual Page Class Key Protection mechanism provides the means to assign virtual pages to one of 32 classes, and to modify data access permissions for each class by modifying the Authority Mask Register (AMR), shown in Figure 41, and to modify instruction access permissions for each class by modifying the Instruction Authority Mask Register (IAMR) shown in Figure 42.
Chapter 6. Storage Control

Version 3.1

1209

Authority Mask Register

The access mask for each class defines the access permissions that apply to loads and stores for which the virtual address is translated using a Page Table Entry that contains a Key field value equal to the class number. The access permissions associated with each class are defined as follows, where AMR2n and AMR2n+1 refer to the first and second bits of the access mask corresponding to class number n.

- A store is permitted if AMR2n=0b0; otherwise the store is not permitted.
- A load is permitted if AMR2n+1=0b0; otherwise the load is not permitted.

The AMR can be accessed using either SPR 13 or SPR 29. Access to the AMR using SPR 29 is privileged.

Programming Note

Because the AMR is part of the program context (if address translation is enabled), and because it is desirable for most application programmers not to have to understand the software synchronization requirements for context alterations (or the nuances of address translation and storage protection), operating systems should provide a system library program that application programs can use to modify the AMR.

Instruction Authority Mask Register

The access mask for each class defines the access permissions that apply to instruction fetches for which the virtual address is translated using a Page Table Entry that contains a Key field value equal to the class number. The access permission associated with each class is defined as follows, where IAMR2n+1 refers to the bit of the access mask corresponding to class number n.

- An instruction fetch is permitted if IAMR2n+1=0b0; otherwise the instruction fetch is not permitted.

Bit 0 of each key field is reserved.

Access to the IAMR is privileged.

The Authority Mask Override Register (AMOR) and the User Authority Mask Override Register (UAMOR), shown in Figure 43 and Figure 44 respectively, can be used to restrict modifications of the AMR. Also, the AMOR can be used to restrict modifications of the UAMOR and IAMR. Access to both the AMOR and UAMOR is privileged. The AMOR is a hypervisor resource.

Programming Note

If address translation is disabled for a given access, the access is not affected by the Virtual Page Class Key Protection mechanism even if the access is made in virtual real addressing mode.

Figure 41. Authority Mask Register (AMR)

Figure 42. Instruction Authority Mask Register (IAMR)

Figure 43. Authority Mask Override Register (AMOR)

Figure 44. User Authority Mask Override Register (UAMOR)

The bits of the AMOR and UAMOR are in 1-1 correspondence with the bits of the AMR (i.e., [U]AMORi corresponds to AMRi). The AMOR affects modifications of the AMR and UAMOR in privileged but non-hypervisor state; the UAMOR affects modifications of the AMR in problem state.

Similarly, the odd bits of the AMOR are in 1-1 correspondence with the odd bits of the IAMR (i.e.,
AMOR_{2j+1} corresponds to IAMR_{2j+1}). The AMOR affects modifications of the IAMR in privileged but non-hypervisor state; the IAMR cannot be accessed in problem state.

- When \textit{mtspr} specifying the AMR (using either SPR 13 or SPR 29) or the IAMR is executed in privileged non-hypervisor state, the AMOR is used as a mask that controls which bits of the resulting AMR or IAMR contents come from register RS and which AMR or IAMR bits are not modified.

- Similarly, when \textit{mtspr} specifying the AMR (using SPR 13) is executed in problem state, the UAMOR is used as a mask that controls which bits of the resulting AMR contents come from register RS and which AMR bits are not modified.

- When \textit{mtspr} specifying the UAMOR is executed in privileged non-hypervisor state, the AMOR is ANDed with the contents of register RS and the result is placed into the UAMOR; the AMOR thereby controls which bits of the resulting UAMOR contents come from register RS and which UAMOR bits are set to zero.

A complete description of these effects can be found in the description of the \textit{mtspr} instruction in Section 5.4.4.

Software must ensure that both bits of each even/odd bit pair of the AMOR contain the same value. — i.e., the contents of register RS for \textit{mtspr} specifying the AMOR must be such that (RS)_2n = (RS)_{2n+1} for every n in the range 0:31 — and likewise for the UAMOR. If this requirement is violated for the UAMOR the results of accessing the UAMOR (including implicitly by the hardware as described in the second item of the preceding list) are boundedly undefined; if the requirement is violated for the AMOR the results of accessing the AMOR (including implicitly by the hardware as described in the first and third items of the list) are undefined.

---

**Programming Note**

The preceding requirement permits designs to implement the AMOR and/or UAMOR as 32-bit registers — specifically, to implement only the even-numbered bits (or only the odd-numbered bits) of the register — in a manner such that the reduction, from the architecturally-required 64 bits to 32 bits, is not visible to (correct) software. This implementation technique saves space in the hardware. (A design that uses this technique does the appropriate “fan in/out” when the register is accessed, to provide the appearance, to (correct) software, of supporting all 64 bits of the register.)

Permitting designs to implement the [U]AMOR as 32-bit registers by virtue of the software requirement specified above, rather than by defining the [U]AMOR as 32-bit registers, permits the architecture to be extended in the future to support controlling modification of the “read access” AMR bits (the odd-numbered bits) independently from the “write access” AMR bits (the even-numbered bits), if that proves desirable. If this independent control does prove desirable, the only architecture change would be to eliminate the software requirement.

---

**Programming Note**

When modifying the AMOR and/or UAMOR, the hypervisor should ensure that the two registers are consistent with one another before giving control to a non-hypervisor program. In particular, the hypervisor should ensure that if AMOR_i=0 then UAMOR_i=0, for all i in the range 0:63. (Having AMOR_i=0 and UAMOR_i=1 would permit problem state programs, but not the operating system, to modify AMR bit i.)
The Virtual Page Class Key Protection mechanism replaces the Data Address Compare mechanism that was defined in versions of the architecture that precede Version 2.04 (e.g., the two facilities use some of the same resources, as described below). However, the Virtual Page Class Key Protection mechanism can be used to emulate the Data Address Compare mechanism. Moreover, programs that use the Data Address Compare mechanism can be modified in a manner such that they will work correctly both on implementations that comply with versions of the architecture that precede Version 2.04 (and hence implement the Data Address Compare mechanism) and on implementations that comply with Version 2.04 of the architecture or with any subsequent version (and hence instead implement the Virtual Page Class Key Protection mechanism). The technique takes advantage of the facts that the SPR number for privileged access to the AMR (29) is the same as the SPR number for the Data Address Compare mechanism's ACCR (Address Compare Control Register), that KEY4 occupies the same bit in the PTE as the Data Address Compare mechanism's AC (Address Compare) bit, and that the definition of ACCR62:63 is very similar to the definition of each even-odd pair of AMR bits. The technique is as follows, where PTE1 refers to doubleword 1 of the PTE.

- Set bits 2:3 and 62:63 of SPR 29 (which is either the ACCR or the AMR) to x, where x is the desired 2-bit value for controlling Data Address Compare matches, and set bits 0:1 to 0s.
- Set PTE1_54 (which is either the AC bit or KEY4) to the same value that the AC bit would be set to, and set PTE1_2:3 (which are either RPN bits, that correspond to a real address size larger than the size supported by any implementation that supports the Data Address Compare mechanism, or KEY0:1) and PTE1_52:53 (which are either reserved bits or KEY2:3) to 0s.
- Use PTEKEY values 0 and 1 only for purposes of emulating the Data Address Compare mechanism, except that PTEKEY value 0 may also be used for any virtual pages for which it is desired that the Virtual Page Class Key Protection mechanism permit all accesses. Do not use PTEKEY =31.
- When a Hypervisor Data Storage interrupt occurs, if HDSISR42=1 then ignore the interrupt for Cache Management instructions other than dcbz. (These instructions can cause a virtual page class key protection violation but cannot cause a Data Address Compare match.) Otherwise forward the interrupt to the operating system, which will treat the interrupt as if a Data Address Compare match had occurred. (Note: Cases for which it is undefined whether a Data Address Compare match occurs do not necessarily cause a virtual page class key protection violation.)

(Because privileged software can access the AMR using either SPR 13 or SPR 29, it might seem that, when SPR 13 was added to the architecture (in Version 2.06), SPR 29 should have been removed. SPR 29 is retained for two reasons: first, to avoid requiring privileged software to change to use the newer SPR number; and second, to retain the ability to emulate the Data Address Compare mechanism as described above.)
When address translation is enabled, the Basic Storage Protection mechanism is controlled by the following.

**Programming Note**

An example of the use of the AMOR (and UAMOR) is to support adjuncts (see Section 6.7.4, “Definitions”). The hypervisor could use KEY value j for all data virtual pages that only the adjunct must be able to access. Before dispatching the partition for the first time, the hypervisor would initialize the three registers as follows.

- **AMR**: all 0s except bits 2j and 2j+1, which would contain 1s
- **UAMOR**: all 0s
- **AMOR**: all 1s except bits 2j and 2j+1, which would contain 0s

Before dispatching the adjunct, the hypervisor would set UAMOR to all 0s, and would set the AMR to all 1s except bits 2j and 2j+1, which would be set to 0s. (Because the adjunct would run in problem state, there is no need for the hypervisor to modify the AMOR, and the adjunct cannot modify the UAMOR.) In addition, the hypervisor would prevent the partition from modifying or deleting PTEs that contain translations used by the adjunct.

(It may be desirable to avoid using KEY values 0, 1, and 31 for storage that only the adjunct can access, because these KEY values may be needed by the partition to emulate the Data Address Compare mechanism, as described above. Also, old software, that was written for an implementation that complies with a version of the architecture that precedes Version 2.04 (the version in which virtual page class keys were added), effectively uses KEY 0 for all virtual pages.)

**Programming Note**

Initialization of the UAMOR to all 0s, by the hypervisor before dispatching a partition for the first time, as described in the preceding Programming Note, permits operating systems (in partitions that run in a compatibility mode corresponding to Version 2.06 of the architecture or a subsequent version) to migrate gradually to supporting problem state access to the AMR — specifically, to avoid having to be changed immediately to modify the UAMOR and to save the AMR contents when an interrupt occurs from problem state. Relatedly, having the UAMOR contain all 0s while an application program is running protects old application programs that are “AMR-unaware”. In the absence of programming errors, such application programs would not attempt to read or modify the AMR. However, having the UAMOR contain all 0s protects such programs against modifying the AMR inadvertently.

Permitting an “AMR-unaware” application program to modify the AMR (inadvertently) is potentially harmful for the obvious reasons. (The program might set to 1 an AMR bit corresponding to accesses that are necessary in order for the program to work correctly.) Moreover, even for an operating system that includes support for problem state modification of the AMR, having the UAMOR contain all 0s allows the operating system to avoid saving and restoring the AMR for “AMR-unaware” application programs. Such an operating system would provide a system service program that allows an application program to declare itself to be “AMR-aware” — i.e., potentially to need to modify the AMR. When an application program invokes this service, the operating system would set the UAMOR to the non-zero value appropriate to the access authorities (load and/or store, for one or more key values) that the application program is allowed to modify, and thereafter would save and restore the AMR (and preserve the UAMOR) for this application program. (Having the UAMOR contain all 0s does not prevent an “AMR-unaware” program from reading the AMR, but inadvertent reading of the AMR is likely to be much less harmful than inadvertently modifying it.)

(For partitions that run in a compatibility mode corresponding to a version of the architecture that precedes Version 2.06, the PCR provides sufficient protection to application programs.)

### 6.7.13.2 Basic Storage Protection, Address Translation Enabled

When address translation is enabled, the Basic Storage Protection mechanism is controlled by the following.
MSR<sub>PR</sub>, which distinguishes between supervisor (privileged) state and problem state
K<sub>s</sub> and K<sub>p</sub>, the supervisor (privileged) state and problem state storage key bits in the SLB entry used to translate the effective address
PP, page protection bits 0:2 in the Page Table Entry used to translate the effective address
For instruction fetches only:
- the N (No-execute) value used for the access (see Sections 6.7.8.1 and 6.7.9.2)
- PTE<sub>G</sub>, the G (Guarded) bit in the Page Table Entry used to translate the effective address

Using the above values, the following rules are applied.
1. For an instruction fetch, the access is not permitted if the N value is 1 or if PTE<sub>G</sub>=1.
2. For any access except an instruction fetch that is not permitted by rule 1, a “Key” value is computed using the following formula:

   \[ \text{Key} = (K_p \& \text{MSR}_{PR}) \mid (K_s \& \neg \text{MSR}_{PR}) \]

   Using the computed Key, Figure 45 is applied. An instruction fetch is permitted for any entry in the figure except “no access”. A load is permitted for any entry except “no access”. A store is permitted only for entries with “read/write”.

<table>
<thead>
<tr>
<th>Key</th>
<th>PP</th>
<th>Access Authority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>read/write</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>read/write</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>read/write</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>read only</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>read only</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>no access</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>read only</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>read/write</td>
</tr>
<tr>
<td>1</td>
<td>011</td>
<td>read only</td>
</tr>
<tr>
<td>1</td>
<td>110</td>
<td>no access</td>
</tr>
</tbody>
</table>

All PP encodings not shown above are reserved. The results of using reserved PP encodings are boundedly undefined.

Figure 45. PP bit protection states, address translation enabled

6.7.13.3 Basic Storage Protection, Address Translation Disabled

When address translation is disabled, the Basic Storage Protection mechanism is controlled by MSR<sub>HV</sub>, which (when MSR<sub>PR</sub>=0) distinguishes between hypervisor state and privileged non-hypervisor state (see Chapter 2 and Section 6.7.3, “Ultravisor Real, Hypervisor Real, and Virtual Real Addressing Modes”). The following rules apply:

1. If MSR<sub>HV</sub>=0, access authority is determined as described in Section 6.7.3.3.
2. If MSR<sub>HV</sub>=1, the access is permitted.

6.7.13.4 Radix Tree Translation Storage Protection

For Radix Tree translation, an attempt to fetch instructions from Guarded storage is a storage protection violation. In all other respects, the storage protection mechanism for Radix Tree translation is completely different from what is provided for HPT translation. EAA<sub>1:3</sub> provide control over read, read/write, and execute access if the process has the appropriate privilege. EAA<sub>0</sub>, together with key 0 in the AMR or IAMR, provide three protection configurations for process-scoped translation: (1) a mode that gives equivalent access to privileged and problem state processes, (2) a mode that gives access only to problem state, and (3) a mode that gives access only to privileged processes. (Note that privileged includes hypervisor privileged.) For partition-scoped translation, including translation of table entry addresses, either value of EAA<sub>0</sub> permits the access. See Figure 36 and Figure 46 for details. The choice of whether to limit access to problem state for process-scoped protection of privileged read and write is determined by key 0 of the AMR. When bit 0 is 0, the privileged bit in the PTE is ignored for a privileged store. When bit 0 is 1, the privileged bit must be 1 for a privileged store. Similarly when bit 1 is 0, the privileged bit in the PTE is ignored for a privileged load. When bit 1 is 1, the privileged bit must be 1 for a privileged load. The choice of whether to limit access to privileged state for process-scoped protection of execute is determined by key 0 of the IAMR. When bit 0 is 0, the privileged bit in the PTE is ignored for an attempt to execute the instruction in privileged state. When bit 1 is 1, the privileged bit must be 1 to execute the instruction in privileged state.
Secure Memory Protection

When SMFCCTRL\_E=1, Secure Memory Protection is enabled. Each location in main storage has a Secure Memory property mem\_SM. mem\_SM=1 indicates secure memory. mem\_SM=0 indicates ordinary memory. Generally, only secure partitions and the ultravisor may access secure memory for explicit and implicit accesses. The one exception is that the Partition Table is commonly located in secure memory, but may be accessed implicitly as part of the translation process for software running with MSR\_P=0. The granularity and method with which main storage is mapped for the Secure Memory property is implementation specific.

For each kind of access to a host real address that can cause a violation of Basic or Radix Tree Translation Storage Protection, a Secure Memory Protection exception is reported by the same type of interrupt as its Basic or Radix Tree Translation Storage Protection counterpart, except setting [H]DSISR or [H]SRR1 bit 43 instead of 36, as follows. For HPT translation, the exception is reported as an ISI or DSI if the thread is in hypervisor state, or if the thread is in non-hypervisor state when IR or DR is 1 for the appropriate type of access and VPM=0; otherwise as HISI or HDSI. For Radix Tree translation, the exception is reported as an ISI or DSI if effLPID=0; otherwise as HISI or HDSI. The same reporting approach is used for accesses which require translation but for which no Basic Storage Protection exception is possible. This includes accesses to the Segment Table Entry Group and Process Table Entry when HPT translation is in use.

In the preceding cases the host real address for the access is a result of address translation. A Secure Memory Protection exception can also be caused by accesses to a host real address that is not the result of address translation. (Such accesses cannot cause a violation of Basic or Radix Tree Translation Storage Protection.) These additional cases are reported as follows. For a hypervisor real mode access the exception is reported as an ISI or DSI. For a process-scoped radix tree access for effLPID=0 the exception is reported as an ISI or DSI. For a PTEG access the exception is reported as an ISI or DSI if MSRHV PR=0b10; otherwise as HISI or HDSI. For a partition-scoped radix tree access the exception is reported as an HISI or HDSI unless effLPID=0, in which case the exception is reported as an ISI or DSI. These cases also set [H]DSISR or [H]SRR1 bit 43 to 1.

6.8 Storage Control Attributes

This section describes aspects of the storage control attributes that are relevant only to privileged software programmers. The rest of the description of storage
control attributes may be found in Section 1.6 of Book II and subsections.

6.8.1 Guarded Storage

Storage is said to be "well-behaved" if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

Storage is said to be Guarded if any of the following conditions is satisfied:

- MSR bit IR or DR is 1 for instruction fetches or data accesses respectively, or MSR$_{HV}$=0, and either G=1 or Att=0b010 in the relevant Page Table Entry.
- MSR bit IR or DR is 0 for instruction fetches or data accesses respectively, MSR$_{HV}$=1, and the storage is specified by the Hypervisor Real Mode Storage Control facility to be treated as Guarded (see Section 6.7.3.2.1).

In general, storage that is not well-behaved should be Guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a Machine Check.

The following rules apply to in-order execution of Load and Store instructions for which the first byte of the storage operand is in storage that is both Caching Inhibited and Guarded.

- **Load or Store** instruction that causes an atomic access
  
  If any portion of the storage operand has been accessed and an External, Decrementer, Hypervisor Decrementer, Performance Monitor, or Imprecise mode Floating-Point Enabled exception is pending, the instruction completes before the interrupt occurs.

- **Load or Store** instruction that causes an Alignment exception, or that causes a [Hypervisor] Data Storage exception for reasons other than Data Address Watchpoint match.

  The portion of the storage operand that is in Caching Inhibited and Guarded storage is not accessed.  

  (The corresponding rules for instructions that cause a Data Address Watchpoint match are given in Section 9.4.)

6.8.1.1 Out-of-Order Accesses to Guarded Storage

In general, Guarded storage is not accessed out-of-order. The only exceptions to this rule are the following.

**Load Instruction**

If a copy of any byte of the storage operand is in a cache then that byte may be accessed in the cache or in main storage.

**Instruction Fetch**

If MSR$_{HV}$=0b10 then an instruction may be fetched if any of the following conditions are met.

1. The instruction is in a cache. In this case it may be fetched from the cache or from main storage.
2. The instruction is in a real page from which an instruction has previously been fetched, except that if that previous fetch was based on condition 1 then the previously fetched instruction must have been in the instruction cache.
3. The instruction is in the same real page as an instruction that is required by the sequential execution model, or is in the real page immediately following such a page.

**Programming Note**

Software should ensure that only well-behaved storage is copied into a cache, either by accessing as Caching Inhibited (and Guarded) all storage that may not be well-behaved, or by accessing such storage as not Caching Inhibited (but Guarded) and referring only to cache blocks that are well-behaved.

If a real page contains instructions that will be executed when MSR$_{IR}$=0 and MSR$_{HV}$=1, software should ensure that this real page and the next real page contain only well-behaved storage (or that the Hypervisor Real Mode Storage Control facility specifies that this real page is not Guarded).

6.8.2 Storage Control Bits

When the thread is not in hypervisor or ultravisor real addressing mode, each storage access is performed under the control of the Page Table Entry used to translate the effective address. Each Page Table Entry contains storage control bits that specify the presence or absence of the corresponding storage control for all accesses translated by the entry as shown in Figure 47 and Figure 48. In the following description, references to individual WIMG bits apply to the corresponding Radix Att encoding, or to the result of combining the process-scoped and partition-scoped ATT encodings.
(see Section 6.7.10.3), except where otherwise stated or obvious from context.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Storage Control Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>0 - not Write Through Required</td>
</tr>
<tr>
<td></td>
<td>1 - Write Through Required</td>
</tr>
<tr>
<td>I</td>
<td>0 - not Caching Inhibited</td>
</tr>
<tr>
<td></td>
<td>1 - Caching Inhibited</td>
</tr>
<tr>
<td>M2</td>
<td>0 - not Memory Coherence Required</td>
</tr>
<tr>
<td></td>
<td>1 - Memory Coherence Required</td>
</tr>
<tr>
<td>G</td>
<td>0 - not Guarded</td>
</tr>
<tr>
<td></td>
<td>1 - Guarded</td>
</tr>
</tbody>
</table>

1 Support for the 1 value of the W bit is optional. Implementations that do not support the 1 value treat the bit as reserved and assume its value to be 0.
2 Support for the 0 value of the M bit is optional, implementations that do not support the 0 value assume the value of the bit to be 1, and may either preserve the value of the bit or write it as 1.

Figure 47. Storage control bits, HPT PTE

<table>
<thead>
<tr>
<th>Att value</th>
<th>Storage Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>normal memory</td>
</tr>
<tr>
<td>01</td>
<td>reserved</td>
</tr>
<tr>
<td>10</td>
<td>non-idempotent I/O</td>
</tr>
<tr>
<td>11</td>
<td>tolerant I/O</td>
</tr>
</tbody>
</table>

W=0 always for Radix Tree translation
M=1 always for Radix Tree translation

Figure 48. Storage control bits, Radix PTE

When the thread is not in hypervisor or ultravisor real addressing mode, instructions are not fetched from storage for which the G bit in the Page Table Entry is set to 1; see Section 6.7.13.

When the thread is in hypervisor or ultravisor real addressing mode, the storage control attributes are implicit; see Section 6.7.3.2.

In Sections 6.8.2.1 and 6.8.2.2, “access” includes accesses that are performed out-of-order, and references to W, I, M, and G bits include the values of those bits that are implied when the thread is in hypervisor or ultravisor real addressing mode.

---

**Programming Note**

In a system consisting of only a single-threaded processor which has caches, correct coherent execution does not require storage to be accessed as Memory Coherence Required, and accessing storage as not Memory Coherence Required may give better performance.

6.8.2.1 Storage Control Bit Restrictions

Process- and partition-scoped Att combinations that specify not Caching Inhibited for the process scope and Caching Inhibited for the partition scope are not permitted. See Table 5.

All combinations of W, I, M, and G values are permitted except those for which both W and I are 1.

---

**Programming Note**

If an application program requests both the Write Through Required and the Caching Inhibited attributes for a given storage location, the operating system should set the I bit to 1 and the W bit to 0.

At any given time, the value of the W bit must be the same for all accesses to a given real page.

At any given time, the value of the I bit must be the same for all accesses to a given real page.

6.8.2.2 Altering the Storage Control Bits

When changing the value of the W bit for a given real page from 0 to 1, software must ensure that no thread modifies any location in the page until after all copies of locations in the page that are considered to be modified in the data caches have been copied to main storage using `dcbst` or `dcbf` (or `dcbstps` or `dcbfps`).

When changing the value of the I bit for a given real page from 0 to 1, software must set the I bit to 1 and then flush all copies of locations in the page from the caches using `dcbf` (or `dcbfps`) and `icbi` before permitting any other accesses to the page. Note that similar cache management is required before using the Fixed-Point Load and Store Caching Inhibited instructions to access storage that has formerly been cached. (See Section 5.4.1 on page 1163.)
When changing the value of the M bit for a given real page, software must ensure that all data caches are consistent with main storage. The actions required to do this are system-dependent.

**Programming Note**

It is recommended that `dcbf` be used, rather than `dcbfl`, when changing the value of the I or W bit from 0 to 1. (`dcbfl` would have to be executed on all threads for which the contents of the data cache may be inconsistent with the new value of the bit, whereas, if the M bit for the page is 1, `dcbf` need be executed on only one thread in the system.)

Additional requirements for changing the storage control bits in the Page Table are given in Section 6.10.
6.9 Storage Control Instructions

6.9.1 Cache Management Instructions

This section describes aspects of cache management that are relevant only to privileged software programmers.

For a `dcbz` instruction that causes the target block to be newly established in the data cache without being fetched from main storage, the hardware need not verify that the associated real address is valid. The existence of a data cache block that is associated with an invalid real address (see Section 6.6) can cause a delayed Machine Check interrupt or a delayed Checkstop.

Each implementation provides an efficient means by which software can ensure that all blocks that are considered to be modified in the data cache have been copied to main storage before the thread enters any power conserving mode in which data cache contents are not maintained.

6.9.2 Synchronize Instruction

The Synchronize instruction is described in Section 4.6.3 of Book II, but only at the level required by an application programmer. This section describes properties of the instruction that are relevant only to operating system, hypervisor, and ultravisor software programmers.

The Synchronize instruction provides an ordering function for stores that are in set A of the memory barrier created by the Synchronize instruction, relative to data accesses caused by instructions that are executed on other threads after the occurrence of the interrupt that is caused by a `msgsndp`, `msgsnd`, or `msgsndu` instruction that follows the Synchronize instruction. The thread that is the target of the `msgsndp`, `msgsnd`, or `msgsndu` instruction is here called the "target thread".

- For `msgsndp`, and \( L = 0, 1, \) or \( 2 \) (or \( 4 \) or \( 5 \)) for the Synchronize instruction, the stores are performed with respect to the target thread before any data accesses caused by instructions that are executed on the target thread after the corresponding Directed Privileged Doorbell interrupt has occurred.

- For `msgsnd` or `msgsndu`, and \( L = 0 \) or \( 2 \) (or \( 4 \)) for the Synchronize instruction, the stores are performed with respect to any given other thread before any data accesses caused by instructions that are executed on the given other thread after a `msgsync` instruction is executed on that thread after the corresponding Directed Hypervisor or Ultravisor Doorbell interrupt has occurred on the target thread.

**Programming Note**

The `msgsync` instruction, which is needed when `msgsnd` or `msgsndu` is used, is not needed when `msgsndp` is used because `msgsndp` targets only threads on the same multi-threaded processor as the thread executing the `msgsndp`, while `msgsnd` and `msgsndu` can target any thread in the system. (If the target thread for `msgsnd` or `msgsndu` is on the same multi-threaded processor as the thread executing the `msgsnd` or `msgsndu`, in principle the `msgsync` can be omitted. This optimization is practical only when the `msgsnd/msgsndu` topology is appropriately constrained, however, because the Directed Hypervisor or Ultravisor Doorbell interrupt provides no indication of which thread executed the `msgsnd` or `msgsndu` that caused the interrupt, so there is no easy way for the interrupt handler to determine whether the `msgsync` can be omitted.) `msgsync` is not needed or defined in V. 2.07 for a similar reason: `msgsnd` in V. 2.07 can target only threads on the same multi-threaded processor as the thread executing the `msgsnd`.

The ordering done by `sync` (and `phwsync` and `ptesync`) provides the appearance of "causality" across a sequence of `msgsnd` (or `msgsndu`) instructions, as in the following example. "msgsnd->T1" means "msgsnd instruction targeting thread T1". "<DHDI 0>" means "occurrence of Directed Hypervisor Doorbell interrupt caused by msgsnd executed on T0". On T0, register r1 is assumed to contain the value 1.

```
| T0          | T1        | T2                          |
| std r1,X    | <DHDI 0>  | <DHDI 1>                    |
| sync        | msgsnd->T2 | msgsync                    |
| msgsnd->T1  |           | ld  r1,X                   |
```

In this example, T2's load from X must return 1.

**Programming Note**

Synchronize with \( L=1 \) (`lwsync`) or \( L=5 \) (`plwsync`) should not be used with `msgsnd` or `msgsndu`. (If used, it will not have the desired ordering effect.)
Another variant of the Synchronize instruction is described below. It is designated the Page Table Entry Synchronize instruction, and is specified by the extended mnemonic \texttt{ptesync} (equivalent to \texttt{sync} with \(L=2\)).

The \texttt{ptesync} instruction has all of the properties of \texttt{sync} with \(L=4\) and also the following additional properties.

- The memory barrier created by the \texttt{ptesync} instruction provides an ordering function for the storage accesses associated with all instructions that are executed by the thread executing the \texttt{ptesync} instruction and, as elements of set A, for all Reference and Change bit updates associated with additional address translations that were performed, by the thread executing the \texttt{ptesync} instruction, before the \texttt{ptesync} instruction is executed. The applicable pairs are all pairs \(a_i,b_j\) in which \(b_j\) is a data access and \(a_i\) is not an instruction fetch.

- The \texttt{ptesync} instruction causes all Reference and Change bit updates associated with address translations that were performed, by the thread executing the \texttt{ptesync} instruction, before the \texttt{ptesync} instruction is executed, to be performed with respect to that thread before the \texttt{ptesync} instruction’s memory barrier is created.

- The memory barrier created by the \texttt{ptesync} instruction provides an ordering function for all stores to the Partition Table, Process Tables, Segment Tables, Page Directories, and Page Tables caused by \texttt{Store} instructions preceding the \texttt{ptesync} instruction with respect to invalidations, of cached copies of information derived from these tables, caused by \texttt{slbieg}, \texttt{slbiag}, and \texttt{tlbie} instructions following the \texttt{ptesync} instruction. The memory barrier ensures that all searches of these tables by another thread, that are performed after an invalidation caused by such an \texttt{slbieg}, \texttt{slbiag}, or \texttt{tlbie} instruction has been performed with respect to the other thread and that implicitly load from the target location of such a store, will obtain the value stored (or a value stored subsequently).

In conjunction with the \texttt{tlbie} and \texttt{tlbsync} instructions, the \texttt{ptesync} instruction provides an ordering function for TLB invalidations and related storage accesses on other threads as described in the \texttt{tlbsync} instruction description on page 1240.

Similarly, in conjunction with the \texttt{slbieg} or \texttt{slbiag} and \texttt{slbsync} instructions, the \texttt{ptesync} instruction provides an ordering function for SLB invalidations and related storage accesses on other threads as described in the \texttt{slbsync} instruction description on page 1230.


delimiter Programming Note

The next bullet is sufficient to order the stores with respect to the invalidations on the thread executing the \texttt{ptesync} instruction. That bullet is also sufficient to provide the ordering with respect to invalidations caused by \texttt{slbie}, \texttt{slbia}, and \texttt{tlbie} instructions, which affect only the thread executing them.

The \texttt{ptesync} instruction provides an ordering function for all stores to the Partition Table, Process Tables, Segment Tables, Page Directories, and Page Tables caused by \texttt{Store} instructions preceding the \texttt{ptesync} instruction with respect to searches of these tables that are performed, by the thread executing the \texttt{ptesync} instruction, after the \texttt{ptesync} instruction completes. Executing a \texttt{ptesync} instruction ensures that all such searches that implicitly load from the target location of such a store will obtain the value stored (or a value stored subsequently). Also, the memory barrier created by the \texttt{ptesync} instruction ensures that all searches of these tables by any other thread, that are performed after a store in set B of the memory barrier has been performed with respect to the other thread and that implicitly load from the target location of such a store, will obtain the value stored (or a value stored subsequently).

Programming Note

For instructions following a \texttt{ptesync} instruction, the memory barrier need not order implicit storage accesses for purposes of address translation and reference and change recording.

The functions performed by the \texttt{ptesync} instruction may take a significant amount of time to complete, so this form of the instruction should be used only if the functions listed above are needed. Otherwise \texttt{sync} with \(L=0\) should be used (or \texttt{sync} with \(L=1\) or with \texttt{SC≠0}, or \texttt{eieio}, or \texttt{sync} with \(L=4\) or \(L=5\) if appropriate).

Section 6.10, “Translation Table Update Synchronization Requirements” on page 1241 gives examples of uses of \texttt{ptesync}.

6.9.3 Lookaside Buffer Management

All implementations have a Segment Lookaside Buffer (SLB). Independent of whether the executing partition operates in a mode that uses hardware SLB loading and bolting versus pure software loading (controlled by the value of LPCR\textsubscript{UPRT}), software is responsible for keeping the SLB current with the segment mapping for the process that is executing. Proper management of the SLB across context switches is described in programming notes.
For performance reasons, most implementations also cache other information that is used in address translation. These caches may include: a Translation Lookaside Buffer (TLB) which is a cache of recently used Page Table Entries (PTEs); a cache of recently used translations of effective addresses to real addresses; a Page Walk Cache for Radix Tree translation; caching of the In-Memory Tables; or any combination of these. Lookaside information, including the SLB, is managed using the instructions described in the subsections of this section unless additional requirements are provided in implementation-specific documentation.

To simplify lookaside buffer management, hardware will only perform speculative translation for the context that is executing, in particular using the current effective values of LPID and PID. Except when LPIDR=0, no translations will be created and cached speculatively when HR=0 and MSRHV=1. Furthermore, no translations will be created and cached speculatively in hypervisor or ultravisor real addressing mode. The limitation of speculative behavior in these situations is to cache a PATE when LPIDR is loaded and a PRTE when PIDR is loaded.

--- Programming Note ---

Speculative Segment Table walks are prohibited when MSRHV=1 and LPIDR=0 because adjacent translations are thread-specific and bolted.

Speculative Segment Table walks are allowed when MSRHV=1 and LPIDR=0 to improve performance for "bare metal" operating systems (operating systems that run in hypervisor state). Bare metal operating systems would use LPIDR=0.

Lookaside information derived from PTEs is not necessarily kept consistent with the Page Table. When software alters the contents of a PTE, in general it must also invalidate all corresponding TLB entries and implementation-specific lookaside information; exceptions to this rule are described in Section 6.10.1.2.

The effects of the slbie, slbieg, slbia, slbiag, and TLB Management instructions on address translations, as specified in Sections 6.9.3.2 for the SLB and 6.9.3.3 for the TLB, Page Walk Cache, and In-Memory Table caches, apply to all implementation-specific lookaside information that is used in address translation. Unless otherwise stated or obvious from context, references to SLB entry invalidation and TLB entry invalidation elsewhere in the Books apply also to invalidation of Page Walk Cache content, In-Memory Table cache content, and all implementation-specific lookaside information that is derived from SLB entries and PTEs, respectively.

All implementations provide a means by which software can invalidate all implementation-specific lookaside information that is derived from PTEs.

Implementation-specific lookaside information that contains translations of effective addresses to real addresses may include "translations" that apply in real addressing mode. Because such "translations" are affected by the contents of the LPCR, HRMOR, and URMOR, when software alters the (relevant) contents of these registers it must also invalidate the corresponding implementation-specific lookaside information. Software can invalidate all such lookaside information by using the slbia instruction with IH=0b0000. However, performance is likely to be better if other, appropriate, IH values are used to limit the amount of lookaside information that is invalidated.

All implementations that have such lookaside information provide a means by which software can invalidate all such lookaside information.

For simplicity, elsewhere in the Books it is assumed that the TLB exists.

--- Programming Note ---

Because the instructions used to manage TLBs, SLBs, Page Walk Caches, caches of Partition and Process Table Entries, and implementation-specific lookaside information may be changed in a future version of the architecture, it is recommended that software "encapsulate" their use into subroutines.

--- Programming Note ---

The function of all the instructions described in Sections 6.9.3.2 - 6.9.3.3 is independent of whether address translation is enabled or disabled.

For a discussion of software synchronization requirements when invalidating SLB and TLB entries, see Chapter 12.

6.9.3.1 Thread-Specific Segment Translations

It is necessary to provide thread-specific temporary ESID to VSID translations. These translations cannot be placed in valid entries in the Segment Table because the Segment Table has a process scope rather than a thread scope. Instead, software will use slbmte to install such translations in the SLB. All SLB entries created using slbmte are considered to be "software created." Software created entries will only translate accesses from the hardware thread by which they are installed. When LPCRUPRT=1, they are also considered to be "bolted." Each thread has the ability to bolt four entries.

6.9.3.2 SLB Management Instructions

The only functionality described in this section that is relevant to Radix Tree translation is the use of slbia to invalidate implementation-specific lookaside informa-
tion. The results of executing any other instruction in this section when HR=1 are boundedly undefined.

Software establishes translations in the SLB using `slb mte`. Care must be taken to avoid creating multiple effective-to-virtual translations for any given effective address. Software-created entries will remain in the SLB until invalidated using `slbie` or `slbia` (which also invalidate related implementation-specific lookaside information) or overwritten using `slb mte`. After updating a Segment Table Entry, software must use an `slbie` or `slbieg` instruction to remove lookaside information associated with the old contents of the entry. `slbie` may be used to invalidate software-created entries, but will not invalidate outboard translation caches. `slbieg` does not invalidate software-created entries, but, together with `slbieg`, is the only way to invalidate outboard translation caches. When taking a PID or LPID out of service with the intent of reusing it, software should use an `slbie` instruction to invalidate the old translation in the SLB, followed by an `slbieg` instruction to invalidate the translation remaining in the lookaside buffers.

### Programming Note

Accesses to a given SLB entry caused by the instructions described in this section obey the sequential execution model with respect to the contents of the entry and with respect to data dependencies on those contents. That is, if an instruction sequence contains two or more of these instructions, when the sequence has completed, the final contents of the SLB entry and of General Purpose Registers is as if the instructions had been executed in program order.

However, software synchronization is required in order to ensure that any alterations of the entry take effect correctly with respect to address translation; see Chapter 12.

### SLB Invalidate Entry X-form

```
<table>
<thead>
<tr>
<th>X-form</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>
```

```
slbie RB
0 31 /// /// RB 434 /
```

<table>
<thead>
<tr>
<th>ea0:35</th>
<th>( (RB)0:35 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{if, for SLB entry that translates} )</td>
<td>( \text{or most recently translated} )</td>
</tr>
<tr>
<td>( \text{EA} )</td>
<td>( \text{ea} ), ( \text{ea} )</td>
</tr>
<tr>
<td>( \text{entry_class} = (RB)36 )</td>
<td>( \text{entry_class} )</td>
</tr>
<tr>
<td>( \text{entry_seg_size} = \text{size specified in} (RB)37:38 )</td>
<td>( \text{entry_seg_size} )</td>
</tr>
<tr>
<td>( \text{then for SLB entry (if any) that translates} )</td>
<td>( \text{then for SLB entry (if any) that translates} )</td>
</tr>
<tr>
<td>( \text{EA} )</td>
<td>( \text{ea} )</td>
</tr>
<tr>
<td>( \text{SLBE}_0 )</td>
<td>( \text{SLBE}_0 )</td>
</tr>
<tr>
<td>( \text{u} )</td>
<td>( \text{u} )</td>
</tr>
<tr>
<td>( \text{undefined 1-bit value} )</td>
<td>( \text{undefined 1-bit value} )</td>
</tr>
<tr>
<td>( \text{if} )</td>
<td>( \text{if} )</td>
</tr>
<tr>
<td>( \text{u} )</td>
<td>( \text{u} )</td>
</tr>
<tr>
<td>( \text{then} )</td>
<td>( \text{then} )</td>
</tr>
<tr>
<td>( \text{if an SLB entry translates} )</td>
<td>( \text{if an SLB entry translates} )</td>
</tr>
<tr>
<td>( \text{esid} )</td>
<td>( \text{esid} )</td>
</tr>
<tr>
<td>( \text{SLBE}_0 )</td>
<td>( \text{SLBE}_0 )</td>
</tr>
<tr>
<td>( \text{v} )</td>
<td>( \text{v} )</td>
</tr>
<tr>
<td>( \text{undefined} )</td>
<td>( \text{undefined} )</td>
</tr>
</tbody>
</table>

The operation performed by this instruction is based on the contents of register RB. The contents of this register are shown below.

```
<table>
<thead>
<tr>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESID</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>RS0:31</th>
<th>PID</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS32:63</td>
<td>LPID</td>
</tr>
<tr>
<td>RB35</td>
<td>ESID</td>
</tr>
<tr>
<td>RB36</td>
<td>Class</td>
</tr>
<tr>
<td>RB37:38</td>
<td>B</td>
</tr>
<tr>
<td>RB39:63</td>
<td>must be 0b0</td>
</tr>
</tbody>
</table>

Let the Effective Address (EA) be any EA for which \( \text{EA}0:35 = (RB)0:35 \). Let the class be \( (RB)36 \). Let the segment size be equal to the segment size specified in \( (RB)37:38 \); the allowed values of \( (RB)37:38 \), and the correspondence between the values and the segment size, are the same as for the B field in the SLBE (see Figure 28 on page 1191).

The class value and segment size must be the same as the class value and segment size in the SLB entry that translates the EA, or the values that were in the SLB entry that most recently translated the EA if the translation is no longer in the SLB; if these values are not the same, it is implementation-dependent whether the SLB entry (or implementation-dependent translation information) that translates the EA is invalidated, and the next paragraph need not apply.

If the SLB contains only a single entry that translates the EA, then that is the only SLB entry that is invalidated, except that it is implementation-dependent whether an implementation-specific lookaside entry for
a real mode address “translation” is invalidated. If the
SLB contains more than one such entry, then zero or
more such entries are invalidated, and similarly for any
implementation-specific lookaside information used in
address translation; additionally, a machine check may
occur.

SLB entries are invalidated by setting the V bit in the
entry to 0, and the remaining fields of the entry are set
to undefined values.

This instruction terminates any Segment Table walks
being performed on behalf of the thread that executes
it.

The hardware ignores the contents of RB listed below
and software must set them to 0s.
- \((RB)_{37}\)
- \((RB)_{39}\)
- \((RB)_{40:63}\)
- If \(s = 40\), \((RB)_{24:35}\)

If this instruction is executed in 32-bit mode, \((RB)_{0:31}\)
must be zeros.

This instruction is privileged.

**Special Registers Altered:**
- None

---

**Programming Note**

The reason the class value specified by `slbie` must
be the same as the Class value that is or was in the
relevant SLB entry is that the hardware may use
these values to optimize invalidation of implementa-
tion-specific lookaside information used in
address translation. If the value specified by `slbie`
differs from the value that is or was in the relevant
SLB entry, these optimizations may produce incor-
rect results. (An example of implementation-spe-
cific address translation lookaside information is
the set of recently used translations of effective
addresses to real addresses that some implement-
tions maintain in an Effective to Real Address
Translation (ERAT) lookaside buffer.) Note that
Radix Tree translations have no defined Class
value, so frequent translation mode transitions may
perform poorly under these optimizations.

When switching tasks in certain cases, it may be
advantageous to preserve some implementa-
tion-specific lookaside entries while invalidating
others. The `slbia` instruction specifying IH value
0b001 or 0b011 can be used for this purpose if SLB
class values are appropriately assigned, i.e., a
class value of 0 indicates that the entry should be
preserved and a class value of 1 indicates the entry
must be invalidated. Also, it is advantageous to
assign a class value of 1 to entries that need to be
invalidated via an `slbie` instruction while preserving
implementation-specific lookaside entries that are
derived from real mode address "translation," SLS
address translation, or translations required to
access the Segment Table Entry Group, since such
entries are assigned a class value of 0.

The B value in register RB may be needed for
invalidating ERAT entries corresponding to the
translation being invalidated.

When switching to execute an adjunct, a hypervisor
will disable translation and use `slbie` to be sure
there is no SLB entry mapping the effective
address space that will be used by the incoming
adjunct. It will then bolt an entry for the incoming
adjunct and transfer control to that adjunct. While
the thread is in hypervisor real addressing mode
and during adjunct execution, no speculative Seg-
ment Table walks will be performed.

---

**SLB Invalidate Entry Global X-form**

<table>
<thead>
<tr>
<th align="left">slbieg</th>
<th align="right">RS, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td align="left">31</td>
<td align="right">RS</td>
</tr>
<tr>
<td align="left">6</td>
<td align="right">///</td>
</tr>
<tr>
<td align="left">11</td>
<td align="right">RB</td>
</tr>
<tr>
<td align="left">16</td>
<td align="right">21</td>
</tr>
<tr>
<td align="left">466</td>
<td align="right">31</td>
</tr>
</tbody>
</table>

---
target_PID = RS0:31
if MSRsvc=1 then target_LPID = RS32:63
else target_LPID = LPIDR

for each thread with LPIDR=target_LPID and PIDR=target_PID
if, for each SLB entry that translates or most recently translated ea
  entry_class = (RB)36 and
  entry_seg_size = size specified in (RB)37:38
then for SLB entry (if any)
  that translates ea and is not software-created
  SLBEV ← 0
  all other fields of SLBE ← undefined
else
  s ← log_base_2(entry_seg_size)
esid ← (RB)0:63-s
u ← undefined 1-bit value
if u then
  if an SLB entry translates esid and the entry
  is not software-created
  SLBEV ← 0
  all other fields of SLBE ← undefined

The operation performed by this instruction is based on the contents of registers RS and RB. The contents of these registers are shown below.

<table>
<thead>
<tr>
<th>RS</th>
<th>PID</th>
<th>LPID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RB</th>
<th>ESID</th>
<th>C</th>
<th>B</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>36</td>
<td>37</td>
<td>39</td>
<td>63</td>
</tr>
</tbody>
</table>

Let the target PID be RS0:31. If the instruction is executed in hypervisor state, let the target LPID be RS32:63; otherwise let the target LPID be the contents of LPIDR. Let the Effective Address (EA) be any EA for which EA0:35 = (RB)0:35. Let the class be (RB)36. Let the segment size be equal to the segment size specified in (RB)37:38, the allowed values of (RB)37:38, and the correspondence between the values and the segment size, are the same as for the B field in the SLBE (see Figure 28 on page 1191).

Only SLBs for threads running on behalf of target_LPID and target_PID are searched. Software-created entries are not invalidated. The class value and segment size must be the same as the class value and segment size in the SLB entry that translates the EA, or the values that were in the SLB entry that most recently translated the EA if the translation is no longer in the SLB; if these values are not the same, it is implementation-dependent whether the SLB entry (or implementation-dependent translation information) that translates the EA is invalidated, and the next paragraph need not apply.

If the SLB contains only a single entry that translates the EA, then that is the only SLB entry that is invalidated, except that it is implementation-dependent whether an implementation-specific lookaside entry for a real mode address “translation” is invalidated. If the SLB contains more than one such entry, then zero or more such entries are invalidated, and similarly for any implementation-specific lookaside information used in address translation; additionally, a machine check may occur.

SLB entries are invalidated by setting the V bit in the entry to 0, and the remaining fields of the entry are set to undefined values.

The hardware ignores the contents of RB listed below and software must set them to 0s.
- (RB)37
- (RB)39
- (RB)40:63
- If s = 40, (RB)24:35
If this instruction is executed in 32-bit mode, (RB)0:31 must be zeros.

The operation performed by this instruction is ordered by the **eieio** (or **p[hi]w[ssyc]n** or **ptesync**) instruction with respect to a subsequent **slbsync** instruction executed by the thread executing the **slbieg** instruction. The operations caused by **slbieg** and **slbsync** are ordered by **eieio** as a fifth set of operations, which is independent of the other four sets that **eieio** orders.

This instruction is privileged except when LPCR_GTSE=0, making it hypervisor privileged.

**Special Registers Altered:**
None

**Programming Note**

**slbieg** does affect SLBs on other threads.
Power ISA™ III

1224

SLB Invalidate All

X-form

**Programming Note**

The reason the class value specified by `slbieg` must be the same as the Class value that is or was in the relevant SLB entry is that the hardware may use these values to optimize invalidation of implementation-specific lookaside information used in address translation. If the value specified by `slbieg` differs from the value that is or was in the relevant SLB entry, these optimizations may produce incorrect results. (An example of implementation-specific address translation lookaside information is the set of recently used translations of effective addresses to real addresses that some implementations maintain in an Effective to Real Address Translation (ERAT) lookaside buffer.) Note that Radix Tree translations have no defined Class value, so frequent translation mode transitions may perform poorly under these optimizations.

When switching tasks in certain cases, it may be advantageous to preserve some implementation-specific lookaside entries while invalidating others. The `slbia` instruction specifying IH value 0b001 or 0b011 can be used for this purpose if SLB class values are appropriately assigned, i.e., a class value of 0 indicates that the entry should be preserved and a class value of 1 indicates the entry must be invalidated. Also, it is advantageous to assign a class value of 1 to entries that need to be invalidated via an `slbieg` instruction while preserving implementation-specific lookaside entries that are derived from real mode address "translation," SLS address translation, or translations required to access the Segment Table Entry Group, since such entries are assigned a class value of 0.

**Programming Note**

The B value in register RB may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

**Programming Note**

Use of `slbieg` to invalidate software-created segment descriptors is a programming error. The architecture requires that bolted entries not be invalidated by the instruction.

```
for each SLB entry except SLB entry 0
  SLBE_v 0
  all other fields of SLBE  undefined

 case (0b011):
  for each SLB entry such that SLBEClass = 1
  SLBE_v 0
  all other fields of SLBE  undefined

 case (0b100):
  for each SLB entry
  SLBE_v 0
  all other fields of SLBE  undefined

 case (0b111):
   slbia invalidates the contents of the SLB, and of implementation-specific lookaside information for effective to real address translations, based on the contents of the IH field as described below. SLB entries are invalidated by setting the V bit in the entry to 0. When an SLB entry is invalidated, the remaining fields of the entry are set to undefined values.

   In the description of the IH values, "implementation-specific lookaside information" is shorthand for "implementation-specific lookaside information for effective to real address translations, based on the contents of the IH field as described below. SLB entries are invalidated by setting the V bit in the entry to 0. When an SLB entry is invalidated, the remaining fields of the entry are set to undefined values.

   The reason the class value specified by `slbieg` must be the same as the Class value that is or was in the relevant SLB entry is that the hardware may use these values to optimize invalidation of implementation-specific lookaside information used in address translation. If the value specified by `slbieg` differs from the value that is or was in the relevant SLB entry, these optimizations may produce incorrect results. (An example of implementation-specific address translation lookaside information is the set of recently used translations of effective addresses to real addresses that some implementations maintain in an Effective to Real Address Translation (ERAT) lookaside buffer.) Note that Radix Tree translations have no defined Class value, so frequent translation mode transitions may perform poorly under these optimizations.

   When switching tasks in certain cases, it may be advantageous to preserve some implementation-specific lookaside entries while invalidating others. The `slbia` instruction specifying IH value 0b001 or 0b011 can be used for this purpose if SLB class values are appropriately assigned, i.e., a class value of 0 indicates that the entry should be preserved and a class value of 1 indicates the entry must be invalidated. Also, it is advantageous to assign a class value of 1 to entries that need to be invalidated via an `slbieg` instruction while preserving implementation-specific lookaside entries that are derived from real mode address "translation," SLS address translation, or translations required to access the Segment Table Entry Group, since such entries are assigned a class value of 0.

   The B value in register RB may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

   Use of `slbieg` to invalidate software-created segment descriptors is a programming error. The architecture requires that bolted entries not be invalidated by the instruction.

   **Programming Note**

   The B value in register RB may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

   **Programming Note**

   Use of `slbieg` to invalidate software-created segment descriptors is a programming error. The architecture requires that bolted entries not be invalidated by the instruction.
The information is for an SLB-derived translation and has a Class value of 1.
- The information is for a Radix Tree-derived translation for which effPID\ne0.

0b100
All SLB entries are invalidated. All implementation-specific lookaside information is invalidated.

0b110
All SLB entries except entry 0 are invalidated; SLB entry 0 is not invalidated. All implementation-specific lookaside information that satisfies any of the following conditions is invalidated:
- The information is for an SLB-derived or SLS translation.
- The information is for a Radix Tree-derived translation for which effLPID\ne0 or effPID\ne0.
- The information was created when address translation was disabled and MSR_{HV PR} was equal to 0b00.

0b111
No SLB entries are invalidated. All implementation-specific lookaside information is invalidated.

**Programming Note**

In the preceding description, “SLB-derived translation” excludes any SLS translation, since SLS translation does not use segmentation.

All other IH values are reserved. If the IH field contains a reserved value, the set of SLB entries and implementation-specific lookaside information that is invalidated by the instruction is undefined.

When IH=0b000, 0b100, or 0b111, execution of this instruction has the side effect of clearing the storage access history associated with the Hypervisor Real Mode Storage Control facility. See Section 6.7.3.2.1, “Hypervisor Real Mode Storage Control” for more details.

This instruction terminates any Segment Table walks being performed on behalf of the thread that executes it, and ensures that any new table walks will be performed using the current PIDR value.

This instruction is privileged.

**Special Registers Altered:**
None

**Programming Note**

When performing a context switch between processes, an HPT operating system will use \texttt{mtPIDR} followed by \texttt{slbia}. The synchronization of the PID value and termination of outstanding Segment Table walks ensures that SLB will not contain multiple entries mapping the same EA range (i.e. from the former and new PIDs). Note that if this sequence is performed with translation enabled, care must be taken to avoid an implicit branch. (i.e. the same translation(s) for the locations containing the context switch routine must be valid for both processes.)

For the corresponding situation when changing partitions from or to a partition using HPT translation, hypervisor software should get all the affected threads into real mode, execute \texttt{mtLPIDR}, and then perform the \texttt{slbia} on all the affected threads. (If the affected threads were not in real mode, avoiding implicit branches due to the \texttt{mtLPIDR} would be very difficult.)

**Programming Note**

\texttt{slbia} does not affect SLBs on other threads.

**Programming Note**

If \texttt{slbia} is executed when instruction address translation is enabled, software can ensure that attempting to fetch the instruction following the \texttt{slbia} does not cause an Instruction Segment interrupt by placing the \texttt{slbia} and the subsequent instruction in the effective segment mapped by SLB entry 0. (The preceding assumes that no other interrupts occur between executing the \texttt{slbia} and executing the subsequent instruction. It also assumes that IH values other than 0b011 and 0b100 are used.)
SLB Invalidate All Global  X-form

```
slbiag RS, L

0 31 RS  /// L  /// 850 /
 0 6 11 15 16 21 31
```

if \( L=0 \) then target_PID = RS0:31
if MSR\(_{hyp}\)=1 then target_LPID = RS32:63
else target_LPID = LPIDR

for each nest SLB
  for each SLBE with LPID=target_LPID and
    (PID=target_PID | L=1)
    SLBE\(_V\) ← 0
    all other fields of SLBE ← undefined

The operation performed by this instruction is based on the contents of register RS. The contents of this register is shown below.

RS

<table>
<thead>
<tr>
<th>PID</th>
<th>LPID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>63</td>
</tr>
</tbody>
</table>

RS\(_{0:31}\) PID
RS\(_{32:63}\) LPID

If \( L=0 \), let the target PID be RS\(_{0:31}\). If the instruction is executed in hypervisor state, let the target LPID be RS\(_{32:63}\); otherwise let the target LPID be the contents of LPIDR.

All nest SLBs are searched. If \( L=0 \), each SLBE for process PID in partition LPID is invalidated. If \( L=1 \), each SLBE for partition LPID is invalidated.

SLB entries are invalidated by setting the V bit in the entry to 0, and the remaining fields of the entry are set to undefined values.

All implementation specific looksaside information associated with SLB-derived translations for the target LPID || PID (L=0) or for the target LPID (L=1) is invalidated. Additional implementation specific looksaside information may be invalidated.

The operation performed by this instruction is ordered by the \( \text{eieio} \) (or \( \text{p}\text{hwsync} \) or \( \text{p}\text{tesync} \)) instruction with respect to a subsequent \( \text{slbsync} \) instruction executed by the thread executing the \( \text{slbiag} \) instruction. The operations caused by \( \text{slbiag} \) and \( \text{slbsync} \) are ordered by \( \text{eieio} \) as a fifth set of operations, which is independent of the other four sets that \( \text{eieio} \) orders.

This instruction is privileged except when LPCR\(_{GTSE}=0\), making it hypervisor privileged.

Special Registers Altered:
None
Chapter 6. Storage Control

### SLB Move To Entry

**X-form**

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>slbmte</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RS</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>///</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>402</td>
<td>(L)</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

When LPCR\_UPRT=0, this instruction is the sole means for specifying Segment translations to the hardware. When LPCR\_UPRT=1, Segment Table walks populate the SLB, and this instruction is used only to bolt thread-specific Segment translations.

The SLB entry specified by bits 52:63 of register RB is loaded from register RS and from the remainder of register RB. The contents of these registers are interpreted as shown in Figure 49.

**RS**

<table>
<thead>
<tr>
<th>B</th>
<th>VSID</th>
<th>(K_s)</th>
<th>(K_P)</th>
<th>NLC</th>
<th>LP</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>52</td>
<td>57</td>
<td>58</td>
<td>60</td>
<td>63</td>
</tr>
</tbody>
</table>

**RB**

<table>
<thead>
<tr>
<th>ESID</th>
<th>V</th>
<th>0s</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>36</td>
<td>37</td>
<td>52</td>
</tr>
</tbody>
</table>

- **RS\_0:1** \(B\)
- **RS\_2:51** VSID
- **RS\_52** \(K_s\)
- **RS\_53** \(K_P\)
- **RS\_54** N
- **RS\_55** L
- **RS\_56** C
- **RS\_57** must be 0b0
- **RS\_58:59** LP
- **RS\_60:63** must be 0b0000
- **RB\_0:35** ESID
- **RB\_36** V
- **RB\_37:51** must be 0b000 || 0x000
- **RB\_52:63** index, which selects the SLB entry

**Figure 49. GPR contents for slbmte**

On implementations that support a virtual address size of only \(n\) bits, \(n<78\), \((RS)_{2:79-n}\) must be zeros.

When LPCR\_UPRT=1, the value of index must not exceed 3. \((RB)_{52:61}\) are ignored.

High-order bits of \((RB)_{52:63}\) that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

The hardware ignores the contents of RS and RB listed below and software must set them to 0s.

- **(RS)_{57}**
- **(RS)_{50:63}**
- **(RB)_{37:51}**

If this instruction is executed in 32-bit mode, \((RB)_{0:31}\) must be zeros (i.e., the ESID must be in the range 0:15).
This instruction must not be used to load a segment descriptor that is in the Segment Table when LPCR\textsubscript{UPRT}=1, and cannot be used to invalidate the translation contained in an SLB entry.

This instruction is privileged.

**Special Registers Altered:**

None

---

**Programming Note**

The reason `slbmte` must not be used to load segment descriptors that are in the Segment Table is that there could be a race condition with hardware loading the same segment descriptor, resulting in duplicate SLB entries. Software must not allow duplicate SLB entries to be created; see Section 6.7.8.2, “SLB Search”.

The reason `slbmte` cannot be used to invalidate an SLB entry is that it does not necessarily affect implementation-specific address translation lookaside information. `slbie` (or `slbia`) must be used for this purpose.

---

### SLB Move From Entry VSID X-form

<table>
<thead>
<tr>
<th>slbmfev</th>
<th>RT,RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
</tr>
</tbody>
</table>

This instruction is used to read software-loaded SLB entries. When LPCR\textsubscript{UPRT}=0, the entry is specified by bits 52:63 of register RB. When LPCR\textsubscript{UPRT}=1, only the first four entries can be read, so bits 52:61 of register RB are ignored. If the specified entry is valid (V=1), the contents of the B, VSID, K\textsubscript{s}, K\textsubscript{p}, N, L, C, and LP fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 50.

#### RT

<table>
<thead>
<tr>
<th>B</th>
<th>VSID</th>
<th>K\textsubscript{s}K\textsubscript{p}NLC</th>
<th>0</th>
<th>LP</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>52</td>
<td>57</td>
<td>58</td>
<td>60</td>
</tr>
</tbody>
</table>

#### RB

<table>
<thead>
<tr>
<th>0s</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
</tr>
</tbody>
</table>

- RT\textsubscript{0:1} B
- RT\textsubscript{2:51} VSID
- RT\textsubscript{52} K\textsubscript{s}
- RT\textsubscript{53} K\textsubscript{p}
- RT\textsubscript{54} N
- RT\textsubscript{55} L
- RT\textsubscript{56} C
- RT\textsubscript{57} set to 0b0
- RT\textsubscript{58:59} LP
- RT\textsubscript{60:63} set to 0b0000

- RB\textsubscript{0:51} must be 0x0_0000_0000_0000

- RB\textsubscript{52:63} index, which selects the SLB entry

**Figure 50. GPR contents for slbmfev**

On implementations that support a virtual address size of only n bits, n<78, RT\textsubscript{2:79-n} are set to zeros.

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), the contents of register RT are set to 0.

High-order bits of (RB)\textsubscript{52:63} that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

The hardware ignores the contents of RB\textsubscript{0:51}.

This instruction is privileged.

The use of the L field is implementation specific.

**Special Registers Altered:**

None
This instruction is used to read software-loaded SLB entries. When LPCR_UPRT=0, the entry is specified by bits 52:63 of register RB. When LPCR_UPRT=1, only the first four entries can be read, so bits 52:61 of register RB are ignored. If the specified entry is valid (V=1), the contents of the ESID and V fields of the entry are placed into register RT. If LPCR_UPRT=1, the value of the BO field of the entry is also placed into register RT. The contents of these registers are interpreted as shown in Figure 51.

**RT**

<table>
<thead>
<tr>
<th>ESID</th>
<th>V</th>
<th>BO</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>36</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>63</td>
</tr>
</tbody>
</table>

**RB**

<table>
<thead>
<tr>
<th></th>
<th>0s</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>63</td>
</tr>
</tbody>
</table>

**Figure 51. GPR contents for slbfee**

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), the contents of register RT are set to 0.

High-order bits of (RB)52:63 that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

The hardware ignores the contents of RB0:51.

This instruction is privileged.

The use of the L field is implementation specific.

**Special Registers Altered:**

None

The SLB is searched for an entry that matches the effective address specified by register RB. When LPCR_UPRT=1, this instruction is nonfunctional. The search is performed as if it were being performed for purposes of address translation. That is, in order for a given entry to satisfy the search, the entry must be valid (V=1), and (RB)0:63-s must equal SLBE[ESID0:63-s] (where 2^s is the segment size selected by the B field in the entry). If exactly one matching entry is found, the contents of the B, VSID, Ks, Kp, N, L, C, and LP fields of the entry are placed into register RT. If no matching entry is found, register RT is set to 0. If more than one matching entry is found, either one of the matching entries is used, as if it were the only matching entry, or a Machine Check occurs. If a Machine Check occurs, register RT, and CR Field 0 are set to undefined values, and the description below of how this register and this field is set does not apply.

The contents of registers RT and RB are interpreted as shown in Figure 52.

**RT**

<table>
<thead>
<tr>
<th>B</th>
<th>VSID</th>
<th>Ks</th>
<th>Kp</th>
<th>NLC</th>
<th>LP</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>52</td>
<td>57</td>
<td>58</td>
<td>60</td>
<td>63</td>
</tr>
</tbody>
</table>

**RB**

<table>
<thead>
<tr>
<th>ESID</th>
<th>0000</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>36</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 52. GPR contents for slbfee**

If s > 28, RT80-s:51 are set to zeros. On implementations that support a virtual address size of only n bits, n < 78, RT2:79-n are set to zeros.

CR Field 0 is set as follows. j is a 1-bit value that is equal to 0b1 if a matching entry was found. Otherwise, j is 0b0. When LPCR_UPRT=0, j=0b0.
**Programming Note**

When LPCR<sub>UPRT</sub>=0, the hypervisor can use `slbfee` to save the contents of any SLBE that the partition has created to map an ESID that is needed by an adjunct, and later use the saved contents to restore the partition-created SLBE after the adjunct has completed execution. The hypervisor must also use `slbie`, twice, first to invalidate the partition-created mapping and later to invalidate the adjunct's mapping.

When LPCR<sub>UPRT</sub>=1, the partition’s SLBE will be restored from the Segment Table by hardware, on demand, after the second `slbie` has been executed. There is no need for the hypervisor to save and restore the partition’s SLBE and hence no need to use `slbfee`.

When the need for LPCR<sub>UPRT</sub>=0 has ended, `slbfee` may be removed from the architecture. Programs that run with LPCR<sub>UPRT</sub>=1 should not use `slbfee`.

| CR0_LT_GTEQ_SO | = 0b00 || j || XER_SO |
|----------------|-------------------------|
| The hardware ignores the contents of RB<sub>36:38</sub> 40:63. |
| If this instruction is executed in 32-bit mode, (RB)<sub>0:31</sub> must be zeros (i.e., the ESID must be in the range 0-15). |
| This instruction is privileged. |

**Special Registers Altered:**

- CR0

---

<table>
<thead>
<tr>
<th>slbsync</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>31 31 31 31 31</th>
</tr>
</thead>
</table>

The `slbsync` instruction provides an ordering function for the effects of all `slbieg` and `slbiag` instructions executed by the thread executing the `slbsync` instruction, with respect to the memory barrier created by a subsequent `ptesync` instruction executed by the same thread. Executing a `slbsync` instruction ensures that all of the following will occur.

- All SLB invalidations caused by `slbieg` and `slbiag` instructions preceding the `slbsync` instruction will have completed on any other thread before any data accesses caused by instructions following the `ptesync` instruction are performed with respect to that thread.

- All storage accesses by other threads for which the address was translated using the translations being invalidated will have been performed with respect to the thread executing the `ptesync` instruction, to the extent required by the associated Memory Coherence Required attributes, before the `ptesync` instruction’s memory barrier is created.

The operation performed by this instruction is ordered by the `eieio` (or `[p]hwsync or `ptesync`) instruction with respect to preceding `slbieg` and `slbiag` instructions executed by the thread executing the `slbsync` instruction. The operations caused by `slbieg` or `slbiag` and `slbsync` are ordered by `eieio` as a fifth set of operations, which is independent of the other four sets that `eieio` orders.

The `slbsync` instruction may complete before operations caused by `slbieg` or `slbiag` instructions preceding the `slbsync` instruction have been performed.

This instruction is privileged except when LPCR<sub>GTSE</sub>=0, making it hypervisor privileged.

See Section 6.10 for a description of other requirements associated with the use of this instruction.

**Special Registers Altered:**

- None

---

| slbsync | should not be used to synchronize the completion of `slbie`. |
6.9.3.3 TLB Management Instructions

In addition to managing the TLB, `tlbie` and `tlbiel` are also used to manage the Page Walk Cache, In-Memory Table caching, and implementation-specific lookaside information that depends on the values of the PTEs. The parameters described below specify the type of translations to invalidate and the scope of the invalidation to be performed.

Radix Invalidation Control (RIC) specifies whether to invalidate the TLB, the Page Walk Cache, or both together with partition and Process Table caching. The RIC values and functions are as follows.

0 Just invalidate TLB.
1 Invalidate just Page Walk Cache.
2 Invalidate TLB, Page Walk Cache, and any caching of Partition and Process Table Entries.
3 Invalidate a group of translations (just in the TLB).

Process Scoped (PRS) specifies whether the translation(s) to be invalidated are partition scoped or process scoped including, for RIC=2, whether process or Partition Table caching is being invalidated.

0 Invalidate partition-scoped translation(s).
1 Invalidate process-scoped translations.

Radix (R) specifies whether the translations to be invalidated are Radix Tree translations or HPT translations. If the R value is incorrect for the target partition, the results of the operation are boundedly undefined. (R is ignored for invalidates with IS=3 and MSR_WV=1 because they have the potential to target translations for multiple partitions.)

0 Invalidate HPT translation(s).
1 Invalidate Radix Tree translations.

Invalidation Selector (IS) (found in RB) specifies the scope of the context to be invalidated.

0 Invalidate just the target VA.
1 Invalidate matching PID.
2 Invalidate matching LPID.
3 If MSR_WV=1, invalidate all entries, otherwise invalidate matching LPID.

The IS=0 RIC=2 variants of `tlbie` and `tlbiel` perform the same TLB invalidations as the corresponding RIC=0 variants, but in addition invalidate Page Walk Cache Entries and partition or Process Table caching associated with the specified LPID or LPID/PID. When RIC=1 and IS=0, the Page Walk Cache Entries for the specified LPID or LPID/PID are invalidated while leaving the corresponding TLB entries intact. The ability to target an individual Page Walk Cache Entry or the set of entries associated with a given Page Table Entry (i.e. IS=0 for RIC=1 or RIC=2) is not supported by the Power ISA. When RIC=3 and IS=0, `tlbie` invalidates a series of consecutive translations for HPT translation. For IS=0 invalidations of Radix Tree translations, the use of `tlbie[ ]` is limited to translations for quadrant 0.

When reassigning an LPID or PID, after updating the Partition and/or Process Table(s) software must use a `tlbie` instruction to remove lookaside information associated with the old partition or process.

To invalidate TLB entries, software must supply an effective page number for process-scoped Radix Tree translations, a guest real page number for partition-scoped Radix Tree translations, and an abbreviated virtual page number for HPT translations. The RTL, RB illustration, and verbal description for R=1 require the reader to make the appropriate mental substitution for partition-scoped invalidation. Note also that where page size is specified to be a function of L and AP, it may also be a function of L and LP. The architecture allows for three independent sets of page sizes, one for R=1, one for RIC=3 (requires R=0), and one for all other cases. An implementation may choose to have a single set of encodings work consistently between any two or all three states.

**TLB Invalidate Entry X-form**

```
X-form

tlbie RB,RS,RIC,PRS,R

0     31 RS / RIC,PRS,R / RB 306 / 31

IS ← (RB)52:53
if MSR_W=1 then search_LPID=RS32:63
else search_LPID=LPID
switch(IS)
case {0b00}:
  If RIC=0
  if R=0 then
    L ← (RB)63
    if L = 0
      then
        base_pg_size = 4K
        actual_pg_size =
        page size specified in (RB)56:58
        i = 51
      else
        base_pg_size =
        base page size specified in (RB)44:51
        actual_pg_size =
        actual page size specified in (RB)44:51
        b ← log_base_2(base_pg_size)
        p ← log_base_2(actual_pg_size)
        i = max{min(43,63-b),63-p}
        sg_size ← segment size specified in (RB)54:55
        for each thread
          for each TLB entry
            if (entry_VA14:i+14 = (RB)0:i)
```
(entry_base_pg_size = base_pg_size) &
(entry_actual_pg_size = actual_pg_size) &
(entry_LPID = search_LPID) &
(entry_processScoped = 0)

then

if ((L = 0) | (b ≥ 20)) then
TLB entry ← invalid
else
if (entry_VA_{8:77-b} = (RB)_{56:75-b}) then
TLB entry ← invalid
else
actual_pg_size = 
page size specified in (RB)_{56:58}
p ← log_base_2(actual_pg_size)

for each thread
for each TLB entry
if (entry_EA0:i = (RB)0:i) &
(entry_actual_pg_size = actual_pg_size) &
(entry_LPID = search_LPID) &
(entry_processScoped = PRS) &
((PRS = 0) | (entry_PID = (RS)0:31))

then
TLB entry ← invalid
else if RIC=3 then

sg_size ← segment size specified in (RB)_{54:55}
pg_size ← f(GS)
number_of_pgs ← g(GS)
p ← log_base_2(pg_size)
n ← log_base_2(number_of_pgs)
i ← 63-p

VA_{14:14+i} ← (RB)_{0:i-n} || n0
do j=0 to 2^n-1, in binary

for each thread
for each TLB entry
if (entry_VA_{14:14+i} = (VA_{14:14+i}+j)) &
(entry_sg_size = sg_size) &
(entry_base_pg_size = pg_size) &
(entry_actual_pg_size = pg_size) &
(entry_LPID = search_LPID) &
(entry_processScoped = 0)

then
TLB entry ← invalid

case {0b01}:
if RIC=0 | RIC=2 then

for each thread
for each TLB entry
if (entry_LPID=search_LPID)
& (entry_PID=RS0:31)
& (entry_PRS=1)

then
TLB entry ← invalid

if RIC=1 | RIC=2 then
for each thread
invalidate process-scoped radix page walk caching associated with partition search_LPID
else
if RIC=1 | RIC=2 then
for each thread
invalid process-scoped page walk caching associated with partition search_LPID
if (PRS=1) then
for each thread invalidate Process Table caching associated with partition search_LPID

else

if RIC=0 then
for each thread
invalidate all partition-scoped page walk caching associated with partition search_LPID
else

if RIC=1 then
for each thread invalidate process-scoped page walk caching associated with partition search_LPID

if RIC=0 |

if MSR Hv then
for all threads
invalidate all partition-scoped page walk caching
else
for all threads
invalidate all process-scoped page walk caching
if (MSR Hv=0) & (PRS=1) then
for each thread invalidate process-scoped page walk caching associated with partition search_LPID

if RIC=2 then
if MSR Hv then
for all threads
invalidate all Partition Table caching
else
for each thread
invalidate all Partition Table caching

else

if RIC=0 | RIC=2 then
for each thread
invalidate all partition-scoped page walk caching
else
for each thread
invalidate all process-scoped page walk caching
if (MSR Hv=0) & (PRS=1) then
for each thread invalidate process-scoped page walk caching associated with partition search_LPID

if RIC=2 then
if MSR Hv then
for all threads
invalidate all Process Table caching
else
for all threads
invalidate all Process Table caching

if RIC=0 then
for each thread
invalidate all partition-scoped page walk caching
else
for each thread
invalidate all process-scoped page walk caching
if (MSR Hv=0) & (PRS=1) then
for each thread invalidate process-scoped page walk caching associated with partition search_LPID

if RIC=2 then
if MSR Hv then
for all threads
invalidate all Process Table caching
else
for all threads
invalidate all Process Table caching

else

if RIC=0 then
for each thread
invalidate all partition-scoped page walk caching
else
for each thread
invalidate all process-scoped page walk caching
if (MSR Hv=0) & (PRS=1) then
for each thread invalidate process-scoped page walk caching associated with partition search_LPID

if RIC=2 then
if MSR Hv then
for all threads
invalidate all Partition Table caching
else
for all threads
invalidate all Partition Table caching

else

if RIC=0 then
for each thread
invalidate all partition-scoped page walk caching
else
for each thread
invalidate all process-scoped page walk caching
if (MSR Hv=0) & (PRS=1) then
for each thread invalidate process-scoped page walk caching associated with partition search_LPID

if RIC=2 then
if MSR Hv then
for all threads
invalidate all Process Table caching
else
for all threads
invalidate all Process Table caching
invalidate all Process Table caching
if (MSR0=0) & (PRS=1) then
for each thread invalidate Process Table
  caching associated with partition
  search_LPID
The operation performed by this instruction is based on
the contents of registers RS and RB. The contents of
these registers are shown below, where IS is (RB)52:53
and L is (RB)63.

RS:

<table>
<thead>
<tr>
<th>PID</th>
<th>LPID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

RB for R=1 and IS=0b00:

<table>
<thead>
<tr>
<th>EPN</th>
<th>IS</th>
<th>0s</th>
<th>AP</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>54</td>
<td>56</td>
<td>59</td>
<td>63</td>
</tr>
</tbody>
</table>

RB for R=0, IS=0b00, RIC=3, and L=0:

<table>
<thead>
<tr>
<th>AVA</th>
<th>IS</th>
<th>B</th>
<th>AP</th>
<th>0s</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>54</td>
<td>56</td>
<td>59</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

RB for R=0, IS=0b00, RIC=3, and L=1:

<table>
<thead>
<tr>
<th>AVA</th>
<th>LP</th>
<th>IS</th>
<th>B</th>
<th>AVAL</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>52</td>
<td>54</td>
<td>56</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

RB for R=0, IS=0b00, and RIC=3:

<table>
<thead>
<tr>
<th>AVA</th>
<th>IS</th>
<th>B</th>
<th>GS</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>54</td>
<td>56</td>
<td>63</td>
</tr>
</tbody>
</table>

If this instruction is executed in hypervisor state,
RS32:63 contains the partition ID (LPID) of the partition
for which one or more translations are being invalid-
ated. Otherwise, the value in LPIDR is used. The
supported (RS)32:63 values are the same as the LPID
values supported in LPIDR. RS0:31 contains a PID
value. The supported values of RS0:31 are the same as
the PID values supported in PIDR.
The following forms are invalid.

- PRS=1, R=0, and RIC=2 (The only pro-
  cess-scoped HPT caching is of the Process Table.)
- RIC=1 and R=0 (There is no Page Walk Cache for
  HPT translation.)
- RIC=3 and R=1 (Group invalidation is only sup-
  ported for HPT translation.)
The following forms are treated as if the instruction
form were invalid.

- RIC=1 and IS=0 (The architecture does not sup-
  port shootout of individual translations in the
  Page Walk Cache.)
- RIC=2 and IS=0 (RIC is for comprehensive invali-
  dation that is not supported at the level of an indi-
  vidual page.)
- RIC=3 and IS=0 (Group invalidation is only sup-
  ported for individual pages.)
- PRS=0 and IS=1 (Partition-scoped translations
  are not associated with processes.)
- R=0, IS=1, and RIC=2 (HPT translations are not
  associated with processes.)
- R=0, RIC=2, PRS=0, HV=0, and IS=2 or 3 (The
  similar cases with RIC=0 allow the HPT OS to
  invalidate all of its TLB entries. The only incremen-
  tial function of these cases is to invalidate partition
table caching, which the OS is not permitted to do.)
The results of an attempt to invalidate a translation
outside of quadrant 0 for Radix Tree translation (R=1,
RIC=0, PRS=1, IS=0, and EA0:1≠0b00) are boundedly
undefined.

IS field in RB contains 0b00
If RIC=0, this is a search for a single TLB entry. The
following relationships must be true and tests and
actions are performed to search for an HPT translation.

If the base page size specified by the PTE that was
used to create the TLB entry to be invalidated is 4
KB, the L field in register RB must contain 0.

If the L field in RB contains 0, the base page size is
4 KB and RB56:58 (AP - Actual Page size field)
must be set to the SLBEI field encoding for the page
size corresponding to the actual page size speci-
fied by the PTE that was used to create the TLB
entry to be invalidated. Thus, b is equal to 12 and p
is equal to log2 (actual page size specified by
(RB)56:58). The Abbreviated Virtual Address (AVA)
field in register RB must contain bits 14:65 of the
virtual address translated by the TLB entry to be
invalidated. Variable i is equal to 51.

If the L field in RB contains 1, the following rules
apply.
The base page size and actual page size are
specified in the LP field in register RB, where
the relationship between (RB)34:51 (LP - Large
Page size selector field) and the base page
size and actual page size is the same as the
relationship between PTELP and the base
page size and actual page size (see Section 6.7.9.1 on page 1195 and Figure 33.
on page 1196). Thus, b is equal to \(\log_2\) (base page size specified by \((RB)_{44:51}\)) and \(p\) is equal to \(\log_2\) (actual page size specified by \((RB)_{44:51}\)). Specifically, \((RB)_{44+c:51}\) must be equal to the contents of bits \(c:7\) of the LP field of the PTE that was used to create the TLB entry to be invalidated, where \(c\) is the number of \("r"\) bits in the LP field of the PTE that was used to create the TLB entry to be invalidated.

Variable \(i\) is the larger of \((63-p)\) and the value that is the smaller of \(43\) and \((63-b)\). \((RB)_{0:i}\) must contain bits \(14:(i+14)\) of the virtual address translated by the TLB to be invalidated. If \(b>20\), \((RB)_{56:43}\) may contain any value and are ignored by the hardware.

If \(b<20\), \((RB)_{56:75-b}\) must contain bits \(58:77-b\) of the virtual address translated by the TLB to be invalidated, and other bits in \((RB)_{56:62}\) may contain any value and are ignored by the hardware.

If \(b\geq20\), \((RB)_{56:62}\) (AVAL - Abbreviated Virtual Address, Lower) may contain any value and are ignored by the hardware.

Let the segment size be equal to the segment size specified in \((RB)_{54:55}\) (B field). The contents of \((RB)_{54:55}\) must be the same as the contents of the B field of the PTE that was used to create the TLB entry to be invalidated.

\(RB_{52:53}\) and \(RB_{59:62}\) (when \((RB)_{63} = 0\)) must contain zeros and are ignored by the hardware.

All TLB entries on all threads that have all of the following properties are made invalid.

- The entry translates a virtual address for which any of the following conditions are met for the partition ID of the partition for which the translation is to be invalidated.
  - The IS field in RB contains 0b10 or MSR HV=0 and \(RIC=0\) or \(RIC=2\).
  - The IS field in RB contains 0b11, and TLBELPID matches the partition ID of the partition for which the translation is to be invalidated.

- The process ID specified in RS matches the process ID in the TLB entry if not invalidating a partition-scoped translation.
- TLBELPID matches the partition ID of the partition for which the translation is to be invalidated.

Additional TLB entries may also be made invalid if those TLB entries contain an LPID that matches the partition ID of the partition for which the translation is to be invalidated.

If \(RIC=3\), then the TLB entries mapping an aligned sequence of virtual pages are made invalid on all threads. The number of virtual pages in the sequence, and their page size (base page size = actual page size), are provided using an implementation-specific encoding of the GS field of RB. The number of virtual pages is a power of two. The abbreviated virtual address of the beginning of the sequence is provided by the AVA field of RB with the appropriate number of low-order bits treated as zero to cause the affected region of VA space to be aligned at a multiple of its size. The effect is as if a tlbie instruction with RIC=PRS=R=0 were executed for each virtual page in the sequence, using the supplied contents of RS and RB except using the AVA value corresponding to the virtual page and using the base and actual page size provided by GS.

**IS field in RB is non-zero**

If \(RIC=0\) or \(RIC=2\), all partition-scoped TLB entries when PRS=0 and either MSR HV=1 or R=0, or all process-scoped TLB entries when PRS=1 on all threads for which any of the following conditions are met for the entry are made invalid.

- The IS field in RB contains 0b10 or MSR HV=0 and the IS field contains 0b11, and TLBELPID matches the partition ID of the partition for which the translation is to be invalidated.
The IS field in RB contains 0b01, TLBE_LPID matches the partition ID of the partition for which the translation is to be invalidated, and TLBE_PPID=RS0:31.

- The IS field in RB contains 0b11 and MSR_HV=1.

If RIC=1 or RIC=2, if the following conditions are met, the respective partition-scoped contents when PRS=0 and MSR_HV=1 or process-scoped contents when PRS=1 of the page walk cache are invalidated.

- If the IS field in RB contains 0b10 or if IS contains 0b11 and MSR_HV=0, for all threads, all properly-scoped page walk caching associated with the partition for which the translation is to be invalidated is invalidated.

- If the IS field in RB contains 0b11 and MSR_HV=1, the entire properly-scoped page walk caching for each thread is invalidated.

- If the IS field in RB contains 0b01 (and PRS=1), for all threads, all properly-scoped page walk caching associated with process RS0:31 in the partition for which the translation is to be invalidated is invalidated.

If RIC=2, if the following conditions are met, the respective partition and Process Table caching are invalidated for all threads.

- If the IS field in RB contains 0b01 and PRS=1, for all threads, caching of Process Table Entries for process RS0:31 in the partition for which the translation is to be invalidated is invalidated.

- If the IS field in RB contains 0b10, MSR_HV=1, and PRS=0, for all threads, caching of Partition Tables for the partition for which the translation is to be invalidated is invalidated.

- If the IS field in RB contains 0b10 and PRS=1, for all threads, caching of Process Tables for the partition for which the translation is to be invalidated is invalidated.

- If the IS field in RB contains 0b11, MSR_HV=1, and PRS=0, for all threads, all Partition Table caching is invalidated.

- If the IS field in RB contains 0b11, MSR_HV=1, and PRS=1, for all threads, all Process Table caching is invalidated.

- If the IS field in RB contains 0b11, MSR_HV=0, and PRS=1, for all threads, caching of Process Tables for the partition for which the translation is to be invalidated is invalidated.

When i>40, RB40:i-1 may contain any value and are ignored by the hardware.

For all IS values

For all threads, any implementation specific lookaside information that is based on any TLB entry that would be invalidated by this instruction will also be invalidated.

MSR_SF must be 1 when this instruction is executed; otherwise the results are undefined.

If the value specified in RS0:31, RS32:63, RB54:55 when R=0, RB56:58 when RB63=0, or RB44:51 when RB63=1 is not supported by the implementation, the instruction is treated as if the instruction form were invalid.

The operation performed by this instruction is ordered by the `eieio` (or `phtlbe` or `ptsync`) instruction with respect to a subsequent `tlbsync` instruction executed by the thread executing the `tlbie` instruction. The operations caused by `tlbie` and `tlbsync` are ordered by `eieio` as a fourth set of operations, which is independent of the other four sets that `eieio` orders.

This instruction is privileged except when LPCR_GTSE=0 or when PRS=0 and HR=1, making it hypervisor privileged.

See Section 6.10, “Translation Table Update Synchronization Requirements” for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonic for `tlbie`:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tlbie</code> RB,RS</td>
<td><code>tlbie</code> RB,RS,0,0,0</td>
</tr>
</tbody>
</table>

Programming Note

`tlbie` serves as both a basic and an extended mnemonic. The Assembler will recognize a `tlbie` mnemonic with five operands as the basic form, and a `tlbie` mnemonic with two operands as the extended form. In the extended form the RIC, PRS, and R operands are omitted and assumed to be 0.

Programming Note

For `tlbie` instructions in which (RB)63=0, the AP value in RB is provided to make it easier for the hardware to locate address translations, in lookaside buffers, corresponding to the address translation being invalidated.

For `tlbie` instructions the AP specification is not binary compatible with versions of the architecture that precede Version 2.06. As an example, for an actual page size of 64 KB AP=0b101, whereas software written for an implementation that complies with a version of the architecture that precedes V. 2.06 would have AP=100 since AP was a 1 bit value followed by 0s in RB57:58. If binary compatibility is important, for a 64 KB page software can use AP=0b101 on these earlier implementations since these implementations were required to ignore RB57:58.
For \texttt{tlbie}[l] instructions the AVA and AVAL fields in RB contain different VA bits from those in PTEAVA.

An operating system that uses HPT translation should only use \texttt{tlbie} to invalidate the translation for a specific page when it knows whether VPM is active, and more specifically, what page size is actually in use for the target translation. The address comparison performed by \texttt{tlbie} is not sensitive to whether VPM is active. As a result, the operating system must supply an AVA value that is appropriate for the page size that is in use.

**TLB Invalidate Entry Local X-form**

\texttt{tlbie} \hspace{1cm} RB,RS,RIC,PRS,R

<table>
<thead>
<tr>
<th>31</th>
<th>30:27</th>
<th>26:19</th>
<th>18:16</th>
<th>15:12</th>
<th>11:6</th>
<th>5:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>R</td>
<td>/</td>
<td>RIC</td>
<td>PRS</td>
<td>R</td>
<td>RB</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td></td>
<td>11</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

\begin{align*}
\text{TLB entry} & \rightarrow \text{invalid} \\
\text{else} & \\
\text{if } & \{\text{entry}_{\text{VA}}_{56:75} = (RB)_{56:75}\} & \text{then} \\
\text{TLB entry} & \rightarrow \text{invalid} \\
\text{else} & \\
\text{pg}_{\text{size}} & = \text{page size specified in } (RB)_{56:58} \\
p & \leftarrow \log_{\text{base}_2}(\text{pg}_{\text{size}}) \\
i & = 63-p \\
\text{for each TLB entry} & \\
\text{if } & \{\text{entry}_{\text{EA}}_{56:75} = (RB)_{56:75}\} & \text{&} \\
\{\text{entry}_{\text{pg}}_{\text{size}} = \text{pg}_{\text{size}}\} & \text{&} \\
\{\text{entry}_{\text{PID}} = \text{search}_{\text{LPID}}\} & \text{&} \\
\{\text{entry}_{\text{process}}_{\text{scoped}} = \text{PRS}\} & \text{&} \\
\{\text{PRS} = 0\} & \text{&} \\
\{\text{entry}_{\text{PID}} = (RS)_{0:31}\} & \text{then} \\
\text{TLB entry} & \rightarrow \text{invalid} \\
\text{case (0b01):} & \\
\text{if SET=0 then} & \\
\text{if } & \text{RIC=0} & \text{&} \text{RIC=2} & \text{then} \\
\text{for each TLB entry} & \\
\text{if } & \{\text{entry}_{\text{LPID}} = \text{search}_{\text{LPID}}\} & \text{&} \\
\{\text{entry}_{\text{PID}} = RS_{0:31}\} & \text{then} \\
\text{TLB entry} & \rightarrow \text{invalid} \\
\text{if } & \text{RIC=1} & \text{&} \text{RIC=2} & \text{then} \\
\text{invalidate process-scoped radix page walk} & \\
caching associated with process RS_{0:31} in partition \text{search}_{\text{LPID}} & \\
\text{if } & \text{(RIC=2)} & \text{&} \text{(PRS=1)} & \text{then} \\
\text{invalidate Process Table caching associated with process RS}_{0:31} in partition \text{search}_{\text{LPID}} & \\
\text{case (0b10):} & \\
\text{if SET=0 then} & \\
\text{if } & \text{RIC=0} & \text{&} \text{RIC=2} & \text{then} \\
\text{if } & \text{PRS=0} & \text{&} \text{(MSRV=1)} & \text{then} \\
\text{all partition-scoped TLB entries} & \rightarrow \text{invalid} \\
\text{if } & \text{PRS=1} & \text{then} \\
\text{for each process-scoped TLB entry} & \\
\text{if } & \{\text{entry}_{\text{LPID}} = \text{search}_{\text{LPID}}\} & \text{&} \\
\{\text{entry}_{\text{process}}_{\text{scoped}} = \text{PRS}\} & \text{then} \\
\text{TLB entry} & \rightarrow \text{invalid} \\
\text{if } & \text{RIC=1} & \text{&} \text{RIC=2} & \text{then} \\
\text{if } & \text{PRS=0} & \text{&} \text{(MSRV=1)} & \text{then} \\
\text{invalidate partition-scoped page walk} & \\
caching associated with partition \text{search}_{\text{LPID}} & \\
\text{if } & \text{PRS=1} & \text{then} \\
\text{invalidate process-scoped page walk} & \\
caching associated with partition \text{search}_{\text{LPID}} & \\
\text{if } & \text{RIC=2} & \text{then} \\
\text{if } & \text{PRS=0} & \text{&} \text{(MSRV=1)} & \text{then} \\
\text{invalidate Partition Table caching associated with partition \text{search}_{\text{LPID}} & \\
\text{if } & \text{PRS=1} & \text{then} \\
\text{invalidate Process Table caching associated with partition \text{search}_{\text{LPID}} & \\
\text{case (0b11):} & \\
\text{if SET=0 then} & \\
\text{if } & \text{RIC=0} & \text{&} \text{RIC=2} & \text{then} \\
\text{if } & \text{(MSRV=1)} & \text{then} \\
\text{all partition-scoped TLB entries} & \rightarrow \text{invalid}
\end{align*}
else
  all process-scoped TLB entries ← invalid
  if (MSRHV=0) & (PRS=1) then
    for each process-scoped TLB entry
      if entry_LPID = search_LPID
        then TLB entry ← invalid
  if (MSRHV=0) & (PRS=0) & (R=0) then
    for each partition-scoped TLB entry
      if entry_LPID = search_LPID
        then TLB entry ← invalid
  if RIC=1 | RIC=2 then
    if MSR HV then
      if PRS=0 then
        invalidate all partition-scoped
        page walk caching
      else
        invalidate all process-scoped
        page walk caching
    if (MSR HV=0) & (PRS=1) then
      invalidate process-scoped page walk
      caching associated with partition
      search_LPID
  if RIC=2 then
    if MSR HV then
      if PRS=0 then
        invalidate all Partition Table caching
      else
        invalidate all Process Table caching
    if (MSR HV=0) & (PRS=1) then
      invalidate Process Table caching
      associated with partition search_LPID

The operation performed by this instruction is based on
the contents of registers RS and RB. The contents of these registers are shown below, where IS is (RB)52:53
and L is (RB)63.

RS:

<table>
<thead>
<tr>
<th>PID</th>
<th>///</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

Programming Note

Note that although there is no PID compare for part-
tition-scoped translation, software must still place
the PID in RS when IS=0 or 1. It may be used, for example, in the TLB hash.

RB for R=1 and IS=0b00:

<table>
<thead>
<tr>
<th>EPN</th>
<th>IS</th>
<th>0s</th>
<th>AP</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>54</td>
<td>56</td>
<td>59</td>
</tr>
</tbody>
</table>

RB for R=0, IS=0b00, and L=0:

<table>
<thead>
<tr>
<th>AVA</th>
<th>IS</th>
<th>B</th>
<th>AP</th>
<th>0s</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>54</td>
<td>56</td>
<td>59</td>
<td>63</td>
</tr>
</tbody>
</table>

Programming Note

In versions of the architecture that precede Version 3.1, \texttt{tlbiel} with IS= 1, 2, or 3 invalidated appropriate
entries only in a specific congruence class of the
TLB, specified by SET in register RB. As a result,
software was required to use a \texttt{tlbiel} loop to iterate
through all congruence classes in order to invali-
date the TLB. Software that will not be run on hard-
ware complying with those versions should specify
SET=0 in register RB. The description for \texttt{tlbiel}
specifies SET instead of 0 in register RB to illus-
trate compatibility with software written to run on
hardware complying with those versions.

LPIDR contains the partition ID (LPID) of the partition for
which the translation is being invalidated. RS0:31 con-
tains a PID value. The supported values of RS0:31 are
the same as the PID values supported in PIDR.

The following forms are invalid.

- PRS=1, R=0, and RIC≠2 (The only pro-
cess-scoped HPT caching is of the Process Table.)
- RIC=1 and R=0 (There is no Page Walk Cache for
  HPT translation.)
- RIC=3 (Group invalidation is not supported for
  \texttt{tlbiel}.)

The following forms are treated as though the instruc-
tion form was invalid.

- RIC=1 and IS=0 (The architecture does not sup-
  port shootdown of individual translations in the
  Page Walk Cache.)
- RIC=1 and SET≠0 (PWC invalidation never
  required a loop to iterate across congruence
  classes.)
- RIC=2 and IS=0 (RIC is for comprehensive invali-
dation that is not supported at the level of an indi-
  vidual page.)
- PRS=0 and IS=1 (Partition-scoped translations
  are not associated with processes.)
- R=0, IS=1, and RIC≠2 (HPT translations are not
  associated with processes.)
- R=0, RIC=2, PRS=0, HV=0, and IS=2 or 3 (The
  similar cases with RIC=0 allow the HPT OS to
  invalidate all of its TLB entries. The only incremen-
tal function of these cases is to invalidate partition
table caching, which the OS is not permitted to do.)
The results of an attempt to invalidate a translation outside of quadrant 0 for Radix Tree translation (R=1, RIC=0, PRS=1, IS=0, and EA0:i=0b00) are boundedly undefined.

**IS field in RB contains 0b00**

If RIC=0, this is a search for a single TLB entry. The following relationships must be true and tests and actions are performed to search for an HPT translation.

If the base page size specified by the PTE that was used to create the TLB entry to be invalidated is 4 KB, the L field in register RB must contain 0.

If the L field in RB contains 0, the base page size is 4 KB and RB56:58 (AP - Actual Page size field) must be set to the SLBE_LIP encoding for the page size corresponding to the actual page size specified by the PTE that was used to create the TLB entry to be invalidated. Thus, b is equal to 12 and p is equal to log2 (actual page size specified by RB56:58). The Abbreviated Virtual Address (AVA) field in register RB must contain bits 14:65 of the virtual address translated by the TLB entry to be invalidated. Variable i is equal to 51.

If the L field in RB contains 1, the following rules apply.

- The base page size and actual page size are specified in the LP field in register RB, where the relationship between (RB)44:51 (LP - Large Page size selector field) and the base page size and actual page size is the same as the relationship between PTE_LP and the base page size and actual page size (see Section 6.7.9.1 on page 1195 and Figure 33 on page 1196). Thus, b is equal to log2 (base page size specified by RB44:51) and p is equal to log2 (actual page size specified by RB44:51). Specifically, (RB)44+c:51 must be equal to the contents of bits c:7 of the LP field of the PTE that was used to create the TLB entry to be invalidated, where c is the number of “r” bits in the LP field of the PTE that was used to create the TLB entry to be invalidated.

- Variable i is the larger of (63-p) and the value that is the smaller of 43 and (63-b). (RB)0:i must contain bits 14:(i+14) of the virtual address translated by the TLB to be invalidated. If b>20, RB64:43 may contain any value and are ignored by the hardware.

- If b<20, (RB)56:75-b must contain bits 58:77-b of the virtual address translated by the TLB to be invalidated, and other bits in (RB)56:62 may contain any value and are ignored by the hardware.

- If b≥20, (RB)56:62 (AVAL - Abbreviated Virtual Address, Lower) may contain any value and are ignored by the hardware.

Let the segment size be equal to the segment size specified in (RB)56:55 (B field). The contents of RB56:55 must be the same as the contents of the B field of the PTE that was used to create the TLB entry to be invalidated.

All TLB entries that have all of the following properties are made invalid on the thread executing the `tlbiel` instruction.

- The entry translates a virtual address for which all the following are true.
  - VA14:14+i is equal to (RB)0:i.
  - L=0 or b≥20 or, if L=1 and b<20, VA56:77-b is equal to (RB)56:77-b.
  - The segment size of the entry is the same as the segment size specified in (RB)54:55.
  - Either of the following is true:
    - The L field in RB is 0, the base page size of the entry is 4 KB, and the actual page size of the entry matches the actual page size specified in (RB)56:58.
    - The L field in RB is 1, the base page size of the entry matches the base page size specified in (RB)44:51, and the actual page size of the entry matches the actual page size specified in (RB)44:51.
  - The entry is partition scoped.
  - TLBE_LPID = LPID_LPID.

The following relationships must be true and tests and actions are performed to search for a Radix Tree translation. For a partition-scoped invalidation, references to the effective address are understood to refer to the guest real address.

The page size is encoded in RB56:58 (AP - Actual Page size field). Thus p is equal to log2 (page size specified by RB56:58). The Effective Page Number (EPN) field in register RB must contain the bits 0:i of the effective address translated by the TLB entry to be invalidated. Variable i is equal to 63-p.

The fields shown as zeros must be set to zero and are ignored by the hardware.

All TLB entries that have all of the following properties are made invalid on the thread executing the `tlbiel` instruction...

- The entry translates an effective address for which EA0:i is equal to (RB)0:i.
- The page size of the entry matches the page size specified in (RB)56:58.
- The entry has the appropriate scope (partition or process).
- The process ID specified in RS matches the process ID in the TLB entry if not invalidating a partition-scoped translation.
- TLBE_LPID matches the partition ID of the partition for which the translation is to be invalidated.

**IS field in RB is non-zero**

When SET=0 is specified and either RIC=0 or RIC=2, each partition-scoped entry when PRS=0 and either MSR_HIV=1 or R=0, or each process-scoped entry when
PRS=1 is invalidated if any of the following conditions are met for the entry.
- The IS field in RB contains 0b10, or MSR$\text{HV}=0$ and the IS field contains 0b11, and TLBE$\text{LPID}=\text{LPID}_\text{LPI}$.  
- The IS field in RB contains 0b01, TLBE$\text{LPID}=\text{LPID}_\text{LPI}$, and TLBE$\text{PID}=\text{RS}_\text{R0:31}$.  
- The IS field in RB contains 0b11 and MSR$\text{HV}=1$.

When SET=0 is specified and either RIC=1 or RIC=2, if the following conditions are met, the respective partition-scoped contents when PRS=0 and MSR$\text{HV}=1$ or process-scoped contents when PRS=1 of the page walk cache are invalidated.
- If the IS field in RB contains 0b10 or if IS contains 0b11 and MSR$\text{HV}=0$, all properly-scoped page walk caching associated with partition LPDIR$\text{LPID}$ is invalidated.  
- If the IS field in RB contains 0b11 and MSR$\text{HV}=1$, the entire properly-scoped page walk caching is invalidated.  
- If the IS field in RB contains 0b01 (and PRS=1), all properly-scoped page walk caching associated with process $\text{RS}_\text{R0:31}$ in partition LPDIR$\text{LPID}$ is invalidated.

When SET=0 is specified and RIC=2, if the following conditions are met, the respective partition and Process Table caching are invalidated.
- If the IS field in RB contains 0b01 and PRS=1, caching of Process Table Entries for process $\text{RS}_\text{R0:31}$ in partition LPDIR$\text{LPID}$ is invalidated.  
- If the IS field in RB contains 0b10, MSR$\text{HV}=1$, and PRS=0, caching of Partition Tables for partition LPDIR$\text{LPID}$ is invalidated.  
- If the IS field in RB contains 0b10 and PRS=1, caching of Process Tables for partition LPDIR$\text{LPID}$ is invalidated.  
- If the IS field in RB contains 0b11, MSR$\text{HV}=1$, and PRS=0, all Partition Table caching is invalidated.  
- If the IS field in RB contains 0b11, MSR$\text{HV}=1$, and PRS=1, all Process Table caching is invalidated.  
- If the IS field in RB contains 0b11, MSR$\text{HV}=0$, and PRS=1, caching of Process Tables for partition LPDIR$\text{LPID}$ is invalidated.

For all IS values

Any implementation specific lookaside information that is based on any TLB entry that would be invalidated by this instruction will also be invalidated.

Depending on the variant of the instruction, RB$\text{R0:39}$, RB$\text{R59:62}$, RB$\text{R59:63}$, RB$\text{R54:55}$, and RB$\text{R54:63}$ are the equivalent of reserved fields, should contain 0s, and are ignored by the hardware.  RS$\text{R32:63}$ is always the equivalent of a reserved field, should contain 0s, and is ignored by the hardware.

Only TLB entries, page walk caching, and process and Segment Table caching on the thread executing the \text{tlb}i\text{e}l instruction are affected.

MSR$\text{SF}$ must be 1 when this instruction is executed; otherwise the results are boundedly undefined.

If the value specified in $\text{RS}_\text{R0:31}$, RB$\text{R54:55}$, RB$\text{R56:58}$, or RB$\text{R44:51}$, when it is needed to perform the specified operation, is not supported by the implementation, the instruction is treated as if the instruction form were invalid.

This instruction is privileged except when PRS=0 and HR=1, making it hypervisor privileged.

See Section 6.10, "Translation Table Update Synchronization Requirements" on page 1241 for a description of other requirements associated with the use of this instruction.

**Special Registers Altered:**
None

**Extended Mnemonics:**

Extended mnemonic for \text{tlb}i\text{e}l:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{tlb}i\text{e}l \text{RB}</td>
<td>\text{tlb}i\text{e}l \text{RB},0,0,0,0</td>
</tr>
</tbody>
</table>

\text{tlb}i\text{e}l serves as both a basic and an extended mnemonic. The Assembler will recognize a \text{tlb}i\text{e}l mnemonic with five operands as the basic form, and a \text{tlb}i\text{e}l mnemonic with one operand as the extended form. In the extended form the RS, RIC, PRS, and R operands are omitted and assumed to be 0.

\text{tlb}i\text{e}l may be executed on a given thread even if the sequence \text{tlb}i\text{e}l - \text{eio} - \text{tlbsyc}n - \text{ptescyc} is concurrently being executed on another thread.

See also the Programming Notes with the description of the \text{tlb}i\text{e}l instruction.
The `tlbsync` instruction provides an ordering function for the effects of all `tlbiel` instructions executed by the thread executing the `tlbsync` instruction, with respect to the memory barrier created by a subsequent `ptesync` instruction executed by the same thread. Executing a `tlbsync` instruction ensures that all of the following will occur.

- All TLB invalidations caused by `tlbiel` instructions preceding the `tlbsync` instruction will have completed on any other thread before any data accesses caused by instructions following the `ptesync` instruction are performed with respect to that thread.

- All storage accesses by other threads for which the address was translated using the translations being invalidated, and all Reference and Change bit updates associated with address translations that were performed by other threads using the translations being invalidated, will have been performed with respect to the thread executing the `ptesync` instruction, to the extent required by the associated Memory Coherence Required attributes, before the `ptesync` instruction's memory barrier is created.

The operation performed by this instruction is ordered by the `eieio` (or `[p]hwsync or `ptesync`) instruction with respect to preceding `tlbiel` instructions executed by the thread executing the `tlbsync` instruction. The operations caused by `tlbiel` and `tlbsync` are ordered by `eieio` as a fourth set of operations, which is independent of the other three sets that `eieio` orders.

The `tlbsync` instruction may complete before operations caused by `tlbiel` instructions preceding the `tlbsync` instruction have been performed.

This instruction is privileged except when LPCRGTE=0, making it hypervisor privileged.

See Section 6.10 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

An operating system that uses HPT translation should only use `tlbiel` to invalidate the translation for a specific page when it knows whether VPM is active, and more specifically, what page size is actually in use for the target translation. The address comparison performed by `tlbiel` is not sensitive to whether VPM is active. As a result, the operating system must supply an AVA value that is appropriate for the page size that is in use.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
| b | b | 11 | 16 | 21 | 31 |
```

The `tlbsync` instruction should not be used to synchronize the completion of `tlbiel`.

**TLB Synchronize**

X-form

`tlbsync`

Programming Note

An operating system that uses HPT translation should only use `tlbiel` to invalidate the translation for a specific page when it knows whether VPM is active, and more specifically, what page size is actually in use for the target translation. The address comparison performed by `tlbiel` is not sensitive to whether VPM is active. As a result, the operating system must supply an AVA value that is appropriate for the page size that is in use.
This section describes rules that software must follow when updating the Translation Tables, and includes suggested sequences of operations for some representative cases. The sequences required for other cases may be deduced from the sequences that are provided and from this accompanying description.

In the sequences of operations shown in the following subsections, the Page Table Entry is assumed to be for a virtual page for which the base page size is equal to the actual page size. If these page sizes are different, multiple `tlbi` instructions are needed, one for each PTE corresponding to the virtual page.

In the sequences of operations shown in the following subsections, any alteration of a translation table entry that corresponds to a single line in the sequence is assumed to be done using a `Store` instruction for which the access is atomic. Appropriate modifications must be made to these sequences if this assumption is not satisfied (e.g., if a store doubleword operation is done using two `Store Word` instructions).

Two correctness-related considerations when choosing translation table update sequences are to be safe for multiple asynchronous sources of update (potentially both hardware and software), and to avoid paradoxes that in some cases could show up as multi-hits in the various translation caches. These considerations lead to the simple, contiguous sequences for general case updates that appear later in this section. Good performance is a third consideration that motivates deferring and/or batching invalidations or even omitting synchronization or invalidation from the general case. The viability of these techniques is determined by whether the lack of a single clear state across the system has problematic repercussions. The discussion of atomic Reference and Change bit updates alludes to one such example. (See Section 6.7.12.) Simpler optimizations are illustrated below.

The following are guidelines for safety when multiple sources of asynchronous updates are possible. To interact correctly with hardware that atomically updates Reference and Change bits (as well as with updates from other software threads), software should use atomic updates to modify valid PTEs. Academically speaking, if hardware uses simple loads and stores, software may either use locking and first invalidate the PTE and cached translations, or may attempt to optimize using atomic updates that don't change the values of the bytes containing the Reference and Change bits with the exception of potentially setting those specific bits to 1 or the Reference bit to 0. When modifying only bytes not subject to hardware modification, software may use either locking or atomic updates, subject to the limitations and optimizations described below. The realities of Reference and Change bit placement may severely limit what optimizations are possible when hardware uses normal loads and stores to update those bits.

To simplify verification and avoid paradoxes, non-impactful limitations are placed on translation table update sequence optimizations. One limitation is that software must not have two or more valid overlapping translations at any level of the translation process with different page or segment sizes. This means that one translation must be marked invalid in the translation table and invalidated from any caches prior to instating the second. The other limitation is that software must not have two or more valid translations with different attributes (i.e. WIMG, ATT). The example of I=1 and I=0 is obvious, but in general there is not enough to be gained to attempt to avoid invalidating one attribute setting before establishing another. In both of these cases, the translation cache invalidation may lag indefinitely behind the table entry invalidations and the cache invalidations may be batched, but must precede enabling the new attributes.

To protect software's ability to have reasonable performance, optimizations that hardware must support are also identified. (These optimizations are understood to be limited by the techniques used for hardware and software updates as described above, and by the properties of the table structure itself.) A convention for atomic updates will yield more opportunity than locking. Hardware that does not use atomic updates may limit or eliminate the opportunity for software to optimize. (The table structure for Radix Tree translation will yield more opportunity than the dual PTEG structure of HPT translation.) Access authority downgrades and setting Change bits to zero may be done without first marking the PTE invalid and invalidating the translation caches. The translation cache invalidation may lag the PTE change indefinitely and be done in bulk. Access authority upgrades and setting Reference and Change bits to 1 may be done without any PTE or translation cache invalidation. Software bits may be changed without any PTE or translation cache invalidation. Finally, any complete change to the RPN (non-overlapping with the original value) does not of itself require synchronization (though other changes to the PTE made at the same time might).

In the following examples, when the same type of sequence works for both types of translation, the HPT PTE is shown because it is more complex. In this description, and in references in subsequent subsections to "safe for multithreaded software," the safety is with respect to the risk of one thread overwriting another's update. There may also be concern for the creation of multiple matching translations, e.g. within a PTEG or pair of PTEGs. When the reservation granule is equal to or larger in size than the structure on which mutual exclusion must be ensured (e.g. PTE for Radix
Tree translation but PTEG for HPT translation), multiple entries will also be prevented. (Secondary hash groups will generally not be covered by the same reservation granule as primary hash groups.)

Updates (by software) to the tables are performed only when they are known to be required by the sequential execution model (see Section 6.5). Because address translation for instructions preceding a given Store instruction might cause an interrupt, and thereby prevent the corresponding store from being required by the sequential execution model, address translations for instructions preceding the Store instruction must be performed before the corresponding store is performed. As a result, an update to a translation table need not be preceded by a context synchronizing instruction.

All of the sequences require a context synchronizing operation after the sequence if the new contents of the translation table are to be used for address translations associated with subsequent instructions.

As noted in the description of the Synchronize instruction in Section 4.6.3 of Book II, address translation associated with instructions which occur in program order subsequent to the Synchronize (and this includes the ptesync variant) may be performed prior to the completion of the Synchronize. To ensure that these instructions and data which may have been speculatively fetched are discarded, a context synchronizing operation is required.

**Programming Note**

In many cases this context synchronization will occur naturally; for example, if the sequence is executed within an interrupt handler the rfid, rfscv, hrfid, or urfid instruction that returns from the interrupt handler may provide the required context synchronization.

Translation table entries must not be changed in a manner that causes an implicit branch.

### 6.10.1 Translation Table Updates

TLBs are non-coherent caches of the HTABs and Radix Trees. TLB entries must be invalidated explicitly with one of the TLB Invalidate instructions. SLBs are non-coherent caches of the Segment Tables, SLB entries must be invalidated explicitly with one of the SLB Invalidate instructions. Page Walk Caches are non-coherent caches of the intermediate steps in Radix Tree translation. Non-coherent caching of the Partition and Process Tables is permitted. Provision has been made for the use of the TLB Invalidate instructions to manage the types of caching described in the preceding two sentences at a PID or LPID granularity.

Unsynchronized lookups in the Page, Segment, and when HR=0, Process Tables continue even while they are being modified. (For Partition Table Entries, and for Process Table Entries when HR=1, the process or partition affected must be inactive because the entries do not have valid bits.) With the exceptions previously identified for Segment Table walks (see Section 6.9.3, “Lookaside Buffer Management”), any thread, including a thread on which software is modifying any of the set of tables described in the first sentence, may look in those tables at any time in an attempt to translate an address. When modifying an entry in any of the former set of tables, software must ensure that the table entry’s V bit is 0 if the table entry does not correctly specify its portion of the translation (e.g., if the RPN field is not correct for the current AVA field).

For HPT translation, updates of Reference and Change bits by the hardware are not synchronized with the accesses that cause the updates. When modifying doubleword 1 of a PTE, software must take care to avoid overwriting a hardware update of these bits and to avoid having the value written by a Store instruction overwritten by a hardware update.

The most basic sequence that will achieve proper system synchronization for PTE updates is the following.  
- tlbie instruction(s) specifying the same LPID operand
- slbsync or slbieg
- tlbie
- ptesync

Other instructions may be interleaved among these instructions. Operating system and hypervisor software that updates Page Table Entries should use this sequence.

Operating systems and nested hypervisors are exposed to being interrupted during this sequence. The interrupting hypervisor is responsible for completing the sequence above. In general this will require the hypervisor to include the following sequence in an interrupt handler.

- eieio
- tlbie
- ptesync

This sequence itself may be interrupted by a higher level hypervisor. When returning to the interrupted software, the original sequence will be completed. Hardware must tolerate the result of nested interleaving of these sequences. tlbie and tlbie sync instructions should only be used as part of these sequences.

The corresponding sequence for Segment Table updates uses slbieg in place of tlbie and slbsync in place of tlbie sync. Similarly slbieg and slbsync should only be used as part of these sequences. In circumstances where a hypervisor may be interrupting either a PTE update or a Segment Table update, it must include both tlbie sync and slbsync in its completing sequence, in either order. Hardware must tolerate the result of nested interleaving of these additional sequences.
The PTE sequence is also used to synchronize updates to Partition Table Entries, and to Process Table Entries that do not have valid bits. Mutual exclusion must be added if the update processes are multi-threaded.

On systems consisting of only a single-threaded processor, the eieio and tlbsync or slbsync instructions can be omitted.

The following subsections illustrate sequences that must be used for translation table updates to tables that are subject to concurrent use by hardware (i.e. that have valid bits in their entries). For Partition Table Entries and for Process Table Entries that do not have valid bits, simpler sequences consisting of just the preceding sequences, perhaps with mutual exclusion if the update processes are multithreaded, is sufficient.

--- Programming Note ---

The eieio instruction prevents the reordering of the preceding tlbie, slbibg, or slbibg instructions with respect to the subsequent tlbsync or slbsync instruction. The tlbsync or slbsync instruction and the subsequent ptesync instruction together ensure that all storage accesses for which the address was translated using the translations being invalidated (by the tlbie, slbibg, or slbibg instructions), and all Reference and Change bit updates associated with address translations that were performed using the translations being invalidated, will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the ptesync instruction are performed with respect to that thread or mechanism.

For Page Table update sequences that mark the PTE invalid (see Section 6.10.1.2, “Modifying a Translation Table Entry”), Reference and Change bit updates cease when the sequence is complete. When the PTE is marked invalid using an atomic update and the Store Conditional setting the entry invalid is successful, the Reference and Change bits obtained by the corresponding Load And Reserve instruction are stable/final values.

The sequences of operations shown in the following subsections assume a multi-threaded environment. In an environment consisting of only a single-threaded processor, the tlbsync or slbsync and the eieio that separates the tlbie or slbibg from the tlbsync or slbsync can be omitted. In a multi-threaded environment, when tlbie or slbibg is used instead of tlbie or slbibg in a Page or Segment Table update, the synchronization requirements are the same as when tlbie or slbibg is used in an environment consisting of only a single-threaded processor.

--- Programming Note ---

For all of the sequences shown in the following subsections, if it is necessary to communicate completion of the sequence to software running on another thread, the ptesync instruction at the end of the sequence should be followed by a Store instruction that stores a chosen value to some chosen storage location X. The memory barrier created by the ptesync instruction ensures that if a Load instruction executed by another thread returns the chosen value from location X, all subsequent searches of the Page or Segment Table by the other thread, that implicitly load from the PTE or STE specified by the sequence’s stores, will obtain the values stored (or values stored subsequently). The Load instruction that returns the chosen value should be followed by a context synchronizing instruction in order to ensure that all instructions following the context synchronizing instruction will be fetched and executed using the values stored by the sequence (or values stored subsequently). (These instructions may have been fetched or executed out-of-order using the old contents of the PTE or STE.)

This Note assumes that the Page or Segment Table and location X are in storage that is Memory Coherence Required.

6.10.1.1 Adding a Page Table Entry

This is the simplest Page Table case. The V bit of the old entry is assumed to be 0. The following sequence can be used to create a PTE, maintain a consistent state, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes. A single quadword store would avoid the need for the eieio. A similar sequence may be used to add a new Segment Table Entry. Mutual exclusion with respect to other software threads may be required, but there is no concern for interaction with hardware updates because the entry is invalid until the last store in the sequence.

```
PTE new values
pp  key  B  ARPN  LP  key  R  C  WING N  pp  key  B  ARPN
  // new values
pp   \* order 1st update before 2nd */
eieio  // order 1st update before 2nd */
PTEAVA SW L H V \* new values (V=1)
ptesync  \* order updates before next
Page Table search and before next
data access  */
```
PTE, maintain a consistent state (subject to the limitations described in the introduction to Section 6.10 such as avoiding overlapping translations), ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

The following sequence is to interact correctly with atomic hardware updates. It returns stable Reference and Change bit values for the old translation and is safe for multiheaded software. If the purpose of the sequence is mainly to collect Reference and Change bit values for the old translation and is subject to non-atomic hardware updates, and is equivalent to deleting the STE and then adding a new one.) Mutual exclusion with respect to other software threads may be required. A similar sequence (except using tlbie with RIC=2 and tlbsync) may be used to modify HR=0 Process Table Entries.

General Case(STE)

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the following sequence can be used to modify the STE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the effective address translated by the new entry will use the correct virtual address and associated attributes. (The sequence is much like the general case for a change to a PTE that is subject to non-atomic hardware updates, and is equivalent to deleting the STE and then adding a new one.) Mutual exclusion with respect to other software threads may be required. A similar sequence (except using tlbie with RIC=2 and tlbsync) may be used to modify HR=0 Process Table Entries.

Reseting the Reference Bit (PTE)

If the only change being made to a valid entry is to set the Reference bit to 0, a simpler sequence suffices because the Reference bit need not be maintained exactly. The byte store is exposed to overwriting another change being performed by multithreaded software, so mutual exclusion may be required.

Setting a Reference or Change Bit or Upgrading Access Authority (PTE Subject to Atomic Hardware Updates)

If the only change being made to a valid PTE that is subject to atomic hardware updates is to set the Refer-
ence or Change bit to 1 or to upgrade access authority, a simpler sequence suffices because the translation hardware will refetch the PTE if an access is attempted for which the only problems were reference and/or change bits needing to be set or insufficient access authority. The store is exposed to overwriting another change being performed by multithreaded software, so mutual exclusion may be required. (Note that changing EAA_0 can be both an upgrade and a downgrade, depending on the value of Key0 of the [[I]AMR]. If it is not solely an upgrade, the simpler sequence must not be used.)

PTE_V L SW R C ALT EAA ← new values (V=1)
ptesync /* order update before next Page Table search and before next data access */

Modifying the SW field (PTE)

If the only change being made to a valid entry is to modify the SW field, the following sequence suffices, because the SW field is not used by the hardware (i.e. is not cached in the TLB and has no effect on hardware behavior).

loop: ldarx r1 ← PTE_dwd_0 /* load dwd 0 of PTE */
if V=0 abort, else/*to interact with locking*/
rs157:60 ← new SW value /* replace SW, in r1 */
stdcx. PTE_dwd_0 ← r1 /* store dwd 0 of PTE */
if still reserved (new SW value, other fields unchanged) */
be- loop /* loop if lost reservation */

A ldarx/stdcx., lbarx/stbcx., or lwarx/stwcx. pair (specifying the low-order byte, halfword, or word respectively of doubleword 0 of the PTE) can be used instead of the ldarx/stdcx. pair shown above for HPT translation. The split SW field in the radix PTE cannot be updated with a single smaller atomic update. This sequence interacts correctly with hardware updates and is safe for multithreaded software. A similar sequence (including the possibility of using a smaller atomic update) can be used to update a Segment Table Entry.

Modifying the Effective Address (STE)

If the effective address translated by a valid STE is to be modified and the new effective address hashes to the same STEG as does the old effective address, the following sequence can be used to modify the STE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the effective address translated by the new entry will use the correct virtual address and associated attributes. Mutual exclusion with respect to other software threads may be required. The corresponding change of the virtual address in the PTE for HPT translation can be performed using a similar sequence, interacting correctly with non-atomic hardware table updates, as long as the second doubleword of the PTE is not stored.

STESID,V ← new values (V=1)
ptesync /* order update before slbieg and before next Segment Table search */
slbieg(old_B,old_ESID,old_TA,old_PID,old_LPID) /*invalidate old translation*/
eieio /* order slbieg before slbsync */
slbsync /* order slbieg before ptesync */
ptesync /* order slbieg, slbsync, and update before next data access */
Chapter 7. Interrupts

7.1 Overview

The Power ISA provides an interrupt mechanism to allow the thread to change state as a result of external signals, errors, or unusual conditions arising in the execution of instructions.

System Reset and Machine Check interrupts are not ordered. All other interrupts are ordered such that only one interrupt is reported, and when it is processed (taken) no program state is lost. Since Save/Restore Registers SRR0 and SRR1 are serially reusable resources used by most interrupts, program state may be lost when an unordered interrupt is taken.

7.2 Interrupt Registers

7.2.1 Machine Status Save/Restore Registers

When various interrupts occur, the state of the machine is saved in the Machine Status Save/Restore registers (SRR0 and SRR1). Section 7.5 describes which registers are altered by each interrupt.

SRR0

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

SRR1

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

Figure 53. Save/Restore Registers

SRR1 bits may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation and, for SRR1 bits in the range 33:36, 42:43, and 45:47, they are specified as being set either to 0 or to an undefined value for all interrupts that set SRR1 (including implementation-dependent setting, e.g. by the Machine Check interrupt or by implementation-specific interrupts). SRR144 cannot be treated as reserved, regardless of how it is set by interrupts, because it is used by software, as described in a Programming Note near the end of Section 7.5.9, “Program Interrupt” on page 1272.

7.2.2 Hypervisor Machine Status Save/Restore Registers

When various interrupts occur, the state of the machine is saved in the Hypervisor Machine Status Save/Restore registers (HSRR0 and HSRR1). Section 7.5 describes which registers are altered by each interrupt.

HSRR0

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

HSRR1

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

Figure 54. Hypervisor Save/Restore Registers

HSRR1 bits may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation and, for HSRR1 bits in the range 33:36 and 42:47, they are specified as being set either to 0 or to an undefined value for all interrupts that set HSRR1 (including implementation-dependent setting, e.g. by implementation-specific interrupts).

The HSRR0 and HSRR1 are hypervisor resources; see Chapter 2.

Programming Note

Execution of some instructions, and fetching instructions when MSR_{IR}=1 or MSR_{HV}=0, may have the side effect of modifying HSRR0 and HSRR1; see Section 7.4.4.

7.2.3 Ultravisor Machine Status Save/Restore Registers

When a Directed Ultravisor Doorbell interrupt occurs, the state of the machine is saved in the Ultravisor Machine Status Save/Restore Registers (USRR0 and USRR1).
USRR1 bits may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation and, for USRR1 bits in the range 33:36 and 42:47, they are specified as being set either to 0 or to an undefined value by the Directed Ultravisor Doorbell interrupt.

The USRR0 and USRR1 are ultravisor resources; see Chapter 3.

7.2.4 Access Segment Descriptor Register

The DAR, HDAR, SRR0, and HSRR0 generally provide the EA for storage exceptions. For hypervisor storage interrupts, additional information is often necessary to enable the hypervisor to handle the interrupt. This information is provided in a 64b SPR called the Access Segment Descriptor Register (ASDR). When nested Radix Tree translation is taking place, the ASDR will generally provide the guest real address down to bit 51. (The smallest supported page size is 4k.) When using paravirtualized HPT translation, information from the segment descriptor that was used to perform the effective to virtual translation is provided in the ASDR. For a big segment the values of the bits of the VSID field that are not part of the VSID are undefined. For exceptions that take place when translating the address of the process table entry or segment table entry group, only the VSID will be provided, because those addresses are specified as virtual addresses and the rest of the segment descriptor is implied. Some instances of the Machine Check interrupt may require the ASDR to be set similarly to how it is set for the hypervisor storage interrupts. The ASDR is set independent of the value of UPRT for the partition that is running.

7.2.5 Data Address Register

The Data Address Register (DAR) is a 64-bit register that is set by the Machine Check, Data Storage, Data Segment, and Alignment interrupts; see Sections 7.5.2, 7.5.3, 7.5.4, and 7.5.8. In general, when one of these interrupts occurs the DAR is set to an effective address associated with the storage access that caused the interrupt, with the high-order 32 bits of the DAR set to 0 if the interrupt occurs in 32-bit mode.

7.2.6 Hypervisor Data Address Register

The Hypervisor Data Address Register (HDAR) is a 64-bit register that is set by the Hypervisor Data Storage Interrupt; see Section 7.5.16. In general, when this interrupt occurs, the HDAR is set to an effective address associated with the storage access that caused the interrupt, with the high-order 32 bits of the HDAR set to 0 if the interrupt occurs in 32-bit mode.

7.2.7 Data Storage Interrupt Status Register

The Data Storage Interrupt Status Register (DSISR) is a 32-bit register that is set by the Machine Check, Data Storage, and Data Segment interrupts; see Sections 7.5.2, 7.5.3, and 7.5.4.
Figure 60. Data Storage Interrupt Status Register

DSISR bits may be treated as reserved in a given implementation if they are specified as being set either to 0 or to an undefined value for all interrupts that set the DSISR.

7.2.8 Hypervisor Data Storage Interrupt Status Register

The Hypervisor Data Storage Interrupt Status Register (HDSISR) is a 32-bit register that is set by the Hypervisor Data Storage interrupt. In general, when one of these interrupts occurs the HDSISR is set to indicate the cause of the interrupt.

Figure 61. Hypervisor Data Storage Interrupt Status Register

7.2.9 Hypervisor Emulation Instruction Register

The Hypervisor Emulation Instruction Register (HEIR) is a 64-bit register that is set by the Hypervisor Emulation Assistance interrupt; see Section 7.5.18. When a word instruction causes the interrupt, the image of the instruction that caused the interrupt is loaded into bits 32:63 of the register, and bits 0:31 are set to 0s. When a prefixed instruction causes the interrupt, the image of the instruction that caused the interrupt is loaded into bits 0:63 of the register.

There may be circumstances in which the suffix cannot be loaded, such as when the instruction is located in storage that is Caching Inhibited, or when the value of the suffix corresponds to a Branch instruction, rfebb, a context synchronizing instruction other than isync, or a “Service Processor Attention” instruction.

In such circumstances, bits 32:63 are set to 0s.

Figure 62. Hypervisor Emulation Instruction Register

<table>
<thead>
<tr>
<th>HEIR</th>
<th>32</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Programming Note

An access to the HMER is likely to be very slow. Software should access it sparingly.

7.2.10 Hypervisor Maintenance Exception Register

Each bit in the Hypervisor Maintenance Exception Register (HMER) is associated with one or more causes of the Hypervisor Maintenance exception, and is set when the associated exception(s) occur. If the corresponding bit in the Hypervisor Maintenance Exception Enable Register (HMEER) is set, a Hypervisor Maintenance Interrupt (HMI) may occur. If the thread is in a power-saving mode when the interrupt would have occurred, the thread will exit the power-saving mode; see Section 7.5.19 and Section 4.3.2.

Figure 63. Hypervisor Maintenance Exception Register

<table>
<thead>
<tr>
<th>HMER</th>
<th>32</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 0    | Set to 1 for a Malfunction Alert. |
| 1    | Set to 1 when performance is degraded for thermal reasons. |
| 2    | Set to 1 when thread recovery is invoked. |
| Others | Implementation-specific. |

When the mtspr instruction is executed with the HMER as the encoded Special Purpose Register, the contents of register RS are ANDed with the contents of the HMER and the result is placed into the HMER.

The exception bits in the HMER are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an mthmer instruction.

Programming Note

When all prefixed instructions are made unavailable by the PCR setting, the prefix will be recognized as an illegal word instruction and placed in HEIR\textsubscript{32:63}.

7.2.11 Hypervisor Maintenance Exception Enable Register

The Hypervisor Maintenance Exception Enable Register (HMEER) is a 64-bit register in which each bit enables the corresponding exception in the HMER to cause the Hypervisor Maintenance interrupt, potentially causing exit from power-saving mode; see Section 7.5.19 and Section 4.3.2.

Figure 64. Hypervisor Maintenance Exception Enable Register

<table>
<thead>
<tr>
<th>HMEER</th>
<th>32</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.2.12 Facility Status and Control Register

The Facility Status and Control Register (FSCR) controls the availability of various facilities in problem state and indicates the cause of a Facility Unavailable interrupt.

When the FSCR makes a facility unavailable, attempted usage of the facility in problem state is treated as follows:

- Execution of an instruction causes a Facility Unavailable exception.
- Access of an SPR using mfspr/mtspr causes a Facility Unavailable exception.
- rfebb, rfid, rfscv, hrfid, urfid, and mtmsr[d] instructions have the same effect on bits in system registers as they would if the bits were available. The same is true for mtspr and mfspr unless the preceding item applies.

MMCR0 can also make various components of the Performance Monitor unavailable when accessed in problem state. An access to one of these components when it is unavailable causes a Facility Unavailable exception.

When the PCR makes a facility unavailable in problem state, the facility is treated as not defined in problem state; any Facility Unavailable interrupt that would occur if the facility were not made unavailable by the PCR does not occur.

When a Facility Unavailable interrupt occurs, the unavailable facility that was accessed is indicated in the most-significant byte of the FSCR.

<table>
<thead>
<tr>
<th>IC</th>
<th>Facility Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

Figure 65. Facility Status and Control Register

The contents of the FSCR are specified below.

### Value Meaning

#### 0:7 Interruption Cause (IC)

When a Facility Unavailable interrupt occurs, the IC field contains a binary number indicating the facility for which access was attempted. The values and their meanings are specified below.

- 02 Access to the DSCR at SPR 3
- 03 Access to a Performance Monitor SPR in group A or B when MMCR0|PMCC is set to a value for which the access results in a Facility Unavailable interrupt. (See the definition of MMCR0|PMCC in Section 10.4.4.)
- 04 Execution of a BHRB Instruction
- 07 Access to an Event-Based Branch SPR or execution of an Event-Based Branch instruction
- 08 Access to the Target Address Register
- 0C Execution of scv
- 0D Execution of a prefixed instruction

All other values are reserved.

#### 8:63 Facility Enable (FE)

The FE field controls the availability of various facilities in problem state as specified below.

- Reserved
- 50 Prefixed Instruction
  - 0 Prefixed instructions are not available in problem state.
  - 1 Prefixed instructions are available in problem state unless made unavailable by another register.

- 51 scv instruction
  - 0 The scv instruction is not available.
  - 1 The scv instruction is available.

- 52:54 Reserved
- 55 Target Address Register (TAR)
  - 0 The TAR and bctar instruction are not available in problem state.
  - 1 The TAR and bctar instruction are available in problem state unless made unavailable by another register.

- 56 Event-Based Branch Facility (EBB)
  - 0 The Event-Based Branch facility SPRs and instructions are not available in problem state, and event-based exceptions and branches do not occur.
  - 1 The Event-Based Branch facility SPRs and instructions (see Chapter 6 of Book II) are available in problem state unless made unavailable by another register, and event-based exceptions and branches are allowed to occur if enabled by other registers.

- 57:60 Reserved

### Programming Note

HFSCRs<sub>59:60</sub> are used to control the availability of the Performance Monitor and the BHRB in problem and privileged non-hypervisor states. FSCRs<sub>59:60</sub> are reserved since the availability of the Performance Monitor and BHRB is controlled by MMCR0.

- 61 Data Stream Control Register at SPR 3 (DSCR)
  - 0 SPR 3 is not available in problem state.
Chapter 7. Interrupts

1 SPR 3 is available in problem state unless made unavailable by another register.

62:63 Reserved

--- Programming Note ---

When an OS has set the FSCR such that a facility is unavailable, the OS should either emulate the facility when it is accessed or provide an application interface that requires the application to request use of the facility before it accesses the facility.

7.2.13 Hypervisor Facility Status and Control Register

The Hypervisor Facility Status and Control Register (HFSCR) controls the availability of various facilities in problem and privileged non-hypervisor states, and indicates the cause of a Hypervisor Facility Unavailable interrupt.

When the HFSCR makes a facility unavailable, attempted usage of the facility in problem or privileged non-hypervisor states is treated as follows:

- Execution of an instruction causes a Hypervisor Facility Unavailable exception.
- Access of an SPR using mfspr/mtspr causes a Hypervisor Facility Unavailable exception
- rfebb, rfid, rfscv, hrfd, urfd, and mtmsr[d] instructions have the same effect on bits in system registers as they would if the bits were available. The same is true for mtspr and mfspr unless the preceding item applies.

When the PCR makes a facility unavailable in problem state, the facility is treated as not defined in problem state; any Hypervisor Facility Unavailable interrupt that would occur if the facility were not made unavailable by the PCR does not occur as a result of problem state access. See Section 2.5 for additional information.

When a Hypervisor Facility Unavailable interrupt occurs, the facility that was accessed is indicated in the most-significant byte of the HFSCR.

<table>
<thead>
<tr>
<th>IC</th>
<th>Facility Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:7</td>
<td>Interruption Cause (IC)</td>
</tr>
</tbody>
</table>

Figure 66. Hypervisor Facility Status and Control Register

The contents of the HFSCR are specified below.

Value Meaning

0:7 Interruption Cause (IC)

When a Hypervisor Facility Unavailable interrupt occurs, the IC field contains a binary number indicating the access that was attempted. The values and their meanings are specified below.

00 Access to a Floating Point register or execution of a Floating Point instruction
01 Access to a Vector or VSX register or execution of a Vector or VSX instruction
02 Access to the DSCR at SPRs 3 or 17
03 Read or write access of a Performance Monitor SPR in group A, or read access of a Performance Monitor SPR in group B. (See Section 10.4.1 for a definition of groups A and B.)
04 Execution of a BHRB Instruction
07 Access to an Event-Based Branch SPR or execution of an Event-Based Branch instruction
08 Access to the Target Address Register
09 Access to the stop instruction in privileged non-hypervisor state when one or more of the following conditions exist.
PSSCR_EC=1
PSSCR_ESL=1
PSSCR_MTL>PSSCR_PSLL
PSSCR_RL>PSSCR_PSLL
0A Access to the msgsndp or msgclrp instructions, the TIR or the DPDES Register

All other values are reserved.

--- Programming Note ---

There is no bit in this register controlling the availability of the stop instruction because the availability of stop in privileged non-hypervisor state is controlled by the PSSCR. See Section 4.2.2.

msgsndp instructions and SPRs (MSGP)

0 The msgsndp and msgclrp instructions and the TIR and DPDES registers are not available in privileged non-hypervisor state.
1 The msgsndp and msgclrp instructions and the TIR and DPDES registers are available in privileged non-hypervisor state unless made unavailable by another register.

53 Reserved

Chapter 7. Interrupts     1251
### Target Address Register (TAR)

0 The TAR and `bctar` instruction are not available in problem and privileged non-hypervisor state.

1 The TAR and `bctar` instruction are available in problem and privileged states unless made unavailable by another register.

### Event-Based Branch Facility (EBB)

0 The Event-Based Branch facility SPRs and instructions are not available in problem and privileged non-hypervisor states, and event-based exceptions and branches do not occur.

1 The Event-Based Branch facility SPRs and instructions are available in problem and privileged states unless made unavailable by another register, and event-based exceptions and branches are allowed to occur if enabled by other bits.

### Data Stream Control Register (DSCR)

0 SPR 3 is not available in problem or privileged non-hypervisor states and SPR 17 is not available in privileged non-hypervisor state.

1 SPR 3 is available in problem and privileged states and SPR 17 is available in privileged state unless made unavailable by another register.

### Vector and VSX Facilities (VECVSX)

0 The facilities whose availability is controlled by either MSRVEC or MSRVEX are not available in problem and privileged non-hypervisor states.

1 The facilities whose availability is controlled by either MSRVEC or MSRVEX are available in problem and privileged states unless made unavailable by another register.

### Floating Point Facility (FP)

0 The facilities whose availability is controlled by MSRFP are not available in problem and privileged non-hypervisor states.

1 The facilities whose availability is controlled by MSRFP are available in problem and privileged states unless made unavailable by another register.

---

### Programming Note

The FSCR can be used to determine whether a particular facility is being used by an application, and the HFSCR can be used to determine whether a particular facility is being used by either an application or by an operating system. This is done by disabling the facility initially, and enabling it in the interrupt handler upon first usage. The information about the usage of a particular facility can be used to determine whether that facility’s state must be saved and restored when changing program context.
Programming Note

The following tables summarize the interrupts that occur as a result of accessing the non-privileged Performance Monitor registers in problem state when MMCR0<sub>PMCC</sub>, PCR, and HFSCR are set to various values. (Accesses to privileged Performance Monitor SPRs (SPRs 784-792, 795-798) in problem state result in Privileged Instruction Type Program interrupts.)

<table>
<thead>
<tr>
<th>SPR</th>
<th>#</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMCR2&lt;sup&gt;A&lt;/sup&gt;</td>
<td>769</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HE,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>MMCR2&lt;sup&gt;A&lt;/sup&gt;</td>
<td>770</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HE,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>PMC1</td>
<td>771</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HE,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>PMC1</td>
<td>772</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HE,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>PMC3</td>
<td>773</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HE,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>PMC4</td>
<td>774</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HE,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>PMC5</td>
<td>775</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HE,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>PMC6</td>
<td>776</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HE,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>MMCR0</td>
<td>779</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HE,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPR</th>
<th>#</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIER2&lt;sup&gt;B&lt;/sup&gt;</td>
<td>736</td>
<td>FU&lt;sup&gt;5&lt;/sup&gt;,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>SIER3&lt;sup&gt;B&lt;/sup&gt;</td>
<td>737</td>
<td>FU&lt;sup&gt;5&lt;/sup&gt;,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>MMCR3&lt;sup&gt;B&lt;/sup&gt;</td>
<td>738</td>
<td>FU&lt;sup&gt;5&lt;/sup&gt;,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>SIER&lt;sup&gt;A&lt;/sup&gt;</td>
<td>768</td>
<td>FU&lt;sup&gt;5&lt;/sup&gt;,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>SDAR</td>
<td>780</td>
<td>FU&lt;sup&gt;5&lt;/sup&gt;,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>MMCR1</td>
<td>782</td>
<td>FU&lt;sup&gt;5&lt;/sup&gt;,HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>FU, HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
<td>HU&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

Notes:
1. Terminology:
   - FU: Facility Unavailable interrupt
   - HE: Hypervisor Emulation Assistance interrupt
   - HU: Hypervisor Facility Unavailable interrupt
2. This SPR is read-only, and cannot be written in any privilege state. (See the mtosp instruction description in Section 5.4.4 for additional information.) FU or HU interrupts do not occur regardless of the value of MMCR0<sub>PMCC</sub> or HFSCR<sub>PM</sub>.
3. When the PCR indicates a version of the architecture prior to V 2.07, this SPR is treated as undefined in problem state; no FU or HU interrupts occur regardless of the value of MMCR0<sub>PMCC</sub> or HFSCR<sub>PM</sub>.
4. An HU interrupt occurs if HFSCR<sub>PM</sub>=0 when this SPR is accessed in either problem state or privileged non-hypervisor state.
5. An FU interrupt occurs only if PCR indicates a version of the architecture subsequent to V 3.0 and MMCR0<sub>PMCC</sub>EXT is set.
6. When the PCR indicates a version of the architecture prior to V 3.1, this SPR is treated as undefined in problem state; no FU or HU interrupts occur regardless of the value of MMCR0<sub>PMCC</sub> or HFSCR<sub>PM</sub>.
When an MSR bit makes a facility unavailable, the facility is made unavailable in all privilege states. Examples of this include the Floating Point, Vector, and VSX facilities. The FSCR and HFSCR affect the availability of facilities only in privilege states that are lower than the privilege of the register (FSCR or HFSCR).
7.3 Interrupt Synchronization

When an interrupt occurs, SRR0, HSRR0, or USRR0 is set to point to an instruction such that all preceding instructions have completed execution, no subsequent instruction has begun execution, and the instruction addressed by SRR0, HSRR0, or USRR0 may or may not have completed execution, depending on the interrupt type.

With the exception of System Reset and Machine Check interrupts, all interrupts are context synchronizing as defined in Section 1.5.1. System Reset and Machine Check interrupts are context synchronizing if they are recoverable (i.e., if bit 62 of SRR1 is set to 1 by the interrupt). If a System Reset or Machine Check interrupt is not recoverable (i.e., if bit 62 of SRR1 is set to 0 by the interrupt), it acts like a context synchronizing operation with respect to subsequent instructions. That is, a non-recoverable System Reset or Machine Check interrupt need not satisfy items 1 through 3 of Section 1.5.1, but does satisfy items 4 and 5.

7.4 Interrupt Classes

Interrupts are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are “system-caused” are:

- System Reset
- Machine Check
- External
- Decrementer
- Directed Privileged Doorbell
- Hypervisor Decrementer
- Hypervisor Maintenance
- Hypervisor Virtualization
- Directed Hypervisor Doorbell
- Directed Ultravisor Doorbell
- Performance Monitor

External, Decrementer, Hypervisor Decrementer, Directed Privileged Doorbell, Directed Hypervisor Doorbell, Directed Ultravisor Doorbell, Hypervisor Maintenance, and Hypervisor Virtualization interrupts are maskable interrupts. Therefore, software may delay the generation of these interrupts. System Reset and Machine Check interrupts are not maskable.

“Instruction-caused” interrupts are further divided into two classes, precise and imprecise.

7.4.1 Precise Interrupt

Except for the Imprecise Mode Floating-Point Enabled Exception type Program interrupt, all instruction-caused interrupts are precise.

When the fetching or execution of an instruction causes a precise interrupt, the following conditions exist at the interrupt point:

1. SRR0, HSRR0, and USRR0 addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the interrupt type and status bits.

2. An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing thread.

3. The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have been partially executed, or may have completed, depending on the interrupt type.

4. Architecturally, no subsequent instruction has begun execution.

7.4.2 Imprecise Interrupt

This architecture defines one imprecise interrupt, the Imprecise Mode Floating-Point Enabled Exception type Program interrupt.

When an Imprecise Mode Floating-Point Enabled Exception type Program interrupt occurs, the following conditions exist at the interrupt point:

1. SRR0 addresses either the instruction causing the exception or some instruction following that instruction; see Section 7.5.9, “Program Interrupt” on page 1272.

2. An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing thread.

3. The instruction addressed by SRR0 may appear not to have begun execution (except, in some cases, for causing the interrupt to occur), may have been partially executed, or may have completed; see Section 7.5.9.

4. No instruction following the instruction addressed by SRR0 appears to have begun execution.

All Floating-Point Enabled Exception type Program interrupts are maskable using the MSR bits FE0 and FE1. Although these interrupts are maskable, they differ significantly from the other maskable interrupts in that the masking of these interrupts is usually controlled by the application program, whereas the masking of all other maskable interrupts is controlled by either the operating system or the hypervisor.
7.4.3 Interrupt Processing

Associated with each kind of interrupt is an interrupt vector, which contains the initial sequence of instructions that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the thread’s state in certain registers, identifying the cause of the interrupt in other registers, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt will occur, the following actions are performed. The handling of Machine Check interrupts (see Section 7.5.2) and System Call Vectored interrupts (see Section 7.5.27) differs from the description given below in several respects.

1. SRR0, HSRR0, or USRR0 is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. Bits 33:36 and 42:47 of SRR1, HSRR1, or USRR1 are loaded with information specific to the interrupt type.
3. Bits 0:32, 37:41, and 48:63 of SRR1, HSRR1, or USRR1 are loaded with a copy of the corresponding bits of the MSR.
4. The MSR is set as shown in Figure 67 on page 1261. In particular, MSR bits IR and DR are set as specified by LPCRAIL or LPCRHAIL as appropriate (see Section 2.2), and MSR bit SF is set to 1, selecting 64-bit mode. The new values take effect beginning with the first instruction executed following the interrupt.
5. Instruction fetch and execution resumes, using the new MSR value, at the effective address specific to the interrupt type. These effective addresses are shown in Figure 68 on page 1262. An offset may be applied to get the effective addresses, as specified by LPCRAIL or LPCRHAIL as appropriate (see Section 2.2).

Interrupts do not clear reservations obtained with lbarx, lharx, lwaxr, ldaxr, or lqarx.

---

Programming Note

In general, when an interrupt occurs, the following instructions should be executed by the interrupt handler before dispatching a “new” program on the thread.

- stbcx, sthcx, stwcx, stdcx, or stqcx, to clear the reservation if one is outstanding, to ensure that a lbarx, lharx, lwaxr, ldaxr, or lqarx in the interrupted program is not paired with a stbcx, sthcx, stwcx, stdcx, stqcx, or waitrsy on the “new” program.
- “eieio, tlbsync, slbsync, ptesync,” to complete any outstanding translation table modification sequence and ensure that all storage accesses caused by the interrupted program will be performed with respect to another thread before the program is resumed on that other thread. (If software conventions are such that there is no possibility of a translation table modification sequence being in progress on the thread, a [p]hwsync instruction suffices.)
- isync or rfid, to ensure that the instructions in the “new” program execute in the “new” context.
- cpabort, to clear state from any previous use of the Copy-Paste Facility.
For instruction-caused interrupts, in some cases it may be desirable for the operating system to emulate the instruction that caused the interrupt, while in other cases it may be desirable for the operating system not to emulate the instruction. The following list, while not complete, illustrates criteria by which decisions regarding emulation should be made. The list applies to general execution environments; it does not necessarily apply to special environments such as program debugging, bring-up, etc.

In general, the instruction should be emulated if:

- The interrupt is caused by a condition for which the instruction description (including related material such as the introduction to the section describing the instruction) implies that the instruction works correctly. Example: Alignment interrupt caused by `lmw` for which the storage operand is not aligned, or by `dcbz` for which the storage operand is in storage that is Write Through Required or Caching Inhibited.

- The instruction is an illegal instruction that should appear, to the program executing it, as if it were supported by the implementation. Example: A Hypervisor Emulation Assistance interrupt is caused by an instruction that has been phased out of the architecture but is still used by some programs that the operating system supports.

If the instruction is a Storage Access instruction, the emulation must satisfy the atomicity requirements described in Section 1.4 of Book II.

In general, the instruction should not be emulated if:

- The purpose of the instruction is to cause an interrupt. Example: System Call interrupt caused by `sc`.

- The interrupt is caused by a condition that is stated, in the instruction description, potentially to cause the interrupt. Example: Alignment interrupt caused by `lwarx` for which the storage operand is not aligned.

- The program is attempting to perform a function that it should not be permitted to perform. Example: Data Storage interrupt caused by `lwz` for which the storage operand is in storage that the program should not be permitted to access. (If the function is one that the program should be permitted to perform, the conditions that caused the interrupt should be corrected and the program re-dispatched such that the instruction will be re-executed. Example: Data Storage interrupt caused by `lwz` for which the storage operand is in storage that the program should be permitted to access but for which there currently is no PTE that satisfies the Page Table search.)
7.4.4 Implicit alteration of HSRR0 and HSRR1

Executing some of the more complex instructions may have the side effect of altering the contents of HSRR0 and HSRR1. The instructions listed below are guaranteed not to have this side effect. Any omission of instruction suffixes is significant; e.g., add is listed but add. is excluded.

1. **Branch instructions**
   - \( b[l][a], \) bc\( [l][a], \) bc\( tr[l], \) bc\( ctr[l] \)

2. **Fixed-Point Load and Store Instructions**
   - \( lbz, \) lb\( zx \), \( lhz, \) lh\( zx \), \( lwz, \) lw\( zx \), \( ld, \) ld\( x \), \( std, \) st\( bx \), st\( h, \) st\( hx \), st\( w, \) st\( wx \), st\( d, \) st\( dx \)

Execution of these instructions is guaranteed not to have the side effect of altering HSRR0 and HSRR1 only if the storage operand is aligned and MSR\( _{HV DR} = 0b100. \)

3. **Arithmetic instructions**
   - addi, addis, add, subf, neg

4. **Compare instructions**
5. **Logical and Extend Sign instructions**

    - `ori, oris, xori, xoris, and, or, xor, nand, nor, eqv, andc, orc, extsb, extsh, extsw`

6. **Rotate and Shift instructions**

    - `rldicl, rldicr, rldic, rlwinm, rlcl, rlm, rldimi, rlwimi, sld, slw, srd, srw`

7. **Other instructions**

    - `isync`
    - `rfid, urfid`
    - `hrfid` in hypervisor state
    - `mtspr, mfspr, mtmsrd, mfmsr`

**Programming Note**

Instructions excluded from the list include the following:

- Instructions that set or use XERCA
- Instructions that set XEROV or XERSO
- `andi, andis`, and fixed-point instructions with Rc=1 (Fixed-point instructions with Rc=1 can be replaced by the corresponding instruction with Rc=0 followed by a Compare instruction.)
- All floating-point instructions
- `mftb`

These instructions, and the other excluded instructions, may be implemented with the assistance of the Hypervisor Emulation Assistance interrupt, or of implementation-specific interrupts that modify HSRR0 and HSRR1. The included instructions are guaranteed not to be implemented thus. (The included instructions are sufficiently simple as to be unlikely to need such assistance. Moreover, they are likely to be needed in interrupt handlers before HSRR0 and HSRR1 have been saved or after HSRR0 and HSRR1 have been restored.)

Similarly, fetching instructions may have the side effect of altering the contents of HSRR0 and HSRR1 unless MSR_{HV IR} = 0b10.
7.5 Interrupt Definitions

Figure 67 shows all the types of interrupts and the values assigned to the MSR for each. Figure 68 shows the effective address of the interrupt vector for each interrupt type. (Section 6.7.5 on page 1184 summarizes all architecturally defined uses of effective addresses, including those implied by Figure 68.)

<table>
<thead>
<tr>
<th>Interrupt Type</th>
<th>MSR Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Reset</td>
<td>0 0 0 0 0 0 p 1 t</td>
</tr>
<tr>
<td>Machine Check</td>
<td>0 0 0 0 0 0 0 1 -</td>
</tr>
<tr>
<td>Data Storage</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>Data Segment</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>Instruction Storage</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>Instruction Segment</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>External</td>
<td>r r 0 0 0 0 h - e -</td>
</tr>
<tr>
<td>Alignment</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>Program</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>Floating-Point Unavailable</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>Decrementer</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>Hypervisor Decrementer</td>
<td>r r 0 0 0 0 - - 1 -</td>
</tr>
<tr>
<td>Directed Privileged Doorbell</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>System Call</td>
<td>r r 0 0 0 0 - s u</td>
</tr>
<tr>
<td>Trace</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>Hypervisor Data Storage</td>
<td>r r 0 0 0 0 - - 1 -</td>
</tr>
<tr>
<td>Hypervisor Instruction Storage</td>
<td>r r 0 0 0 0 - - 1 -</td>
</tr>
<tr>
<td>Hypervisor Emulation Assistance</td>
<td>r r 0 0 0 0 - - 1 -</td>
</tr>
<tr>
<td>Hypervisor Maintenance</td>
<td>0 0 0 0 0 0 - - 1 -</td>
</tr>
<tr>
<td>Directed Hypervisor Doorbell</td>
<td>r r 0 0 0 0 - - 1 -</td>
</tr>
<tr>
<td>Hypervisor Virtualization</td>
<td>r r 0 0 0 0 - - 1 -</td>
</tr>
<tr>
<td>Performance Monitor</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>Vector Unavailable</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>VSX Unavailable</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>Facility Unavailable</td>
<td>r r 0 0 0 0 - - -</td>
</tr>
<tr>
<td>Hypervisor Facility Unavailable</td>
<td>r r 0 0 0 0 - - 1 -</td>
</tr>
<tr>
<td>Directed Ultravisor Doorbell</td>
<td>0 0 0 0 0 - - 1 1</td>
</tr>
<tr>
<td>System Call Vectored</td>
<td>r r 0 0 - - - - -</td>
</tr>
</tbody>
</table>
Chapter 7. Interrupts

Figure 67. MSR setting due to interrupt

<table>
<thead>
<tr>
<th>Interrupt Type</th>
<th>MSR Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bit is set to 0</td>
</tr>
<tr>
<td>1</td>
<td>bit is set to 1</td>
</tr>
<tr>
<td></td>
<td>bit is not altered</td>
</tr>
<tr>
<td>r</td>
<td>for interrupts for which LPCRAIL applies, if LPCRAIL=3, set to 1; for interrupts for which LPCRHAIL applies, if LPCRHAIL=1, set to 1; otherwise set to 0</td>
</tr>
<tr>
<td>p</td>
<td>if the interrupt occurred while the thread was in power-saving mode, set to 1; otherwise not altered</td>
</tr>
<tr>
<td>e</td>
<td>if LPES=0, set to 1; otherwise not altered</td>
</tr>
<tr>
<td>h</td>
<td>if LPES=1, set to 0; otherwise not altered</td>
</tr>
<tr>
<td>s</td>
<td>if LEV=1 or LEV=2, set to 1; otherwise not altered</td>
</tr>
<tr>
<td>t</td>
<td>if the interrupt caused exit from a state-losing power-saving mode and SMFCTRL_E=1, set to 1; if the interrupt caused exit from a state-losing power-saving mode and SMFCTRL_E=0, set to 0; otherwise not altered</td>
</tr>
<tr>
<td>u</td>
<td>if SMFCTRL_E=1 and LEV=2, set to 1; otherwise not altered</td>
</tr>
</tbody>
</table>

*Settings for Other Bits*

Bits PR and PMM are set to 0.

The TE field is set to 0b00.

FP, VEC, VSX, and bits 5 and 31 are set to 0.

If the interrupt results in MSR_SHV being equal to 0b11, the LE bit is copied from the UIL_E bit; otherwise, if the interrupt results in MSR_SHV being equal to 0b01, the LE bit is copied from the HILE bit; otherwise the LE bit is copied from the LPCRILE bit.

The SF bit is set to 1.

Reserved bits are set as if written as 0.
7.5.1 System Reset Interrupt

If a System Reset exception causes an interrupt that is not context synchronizing or causes the loss of a Machine Check exception or a Direct External exception, or if the state of the thread has been corrupted, the interrupt is not recoverable.

When the thread is in any power-saving level, a System Reset interrupt occurs when a System Reset exception exists. When the thread is in a power-saving level that was entered when PSSCREC=1, a System Reset interrupt also occurs when any of the following events occurs provided that the event is enabled to cause exit from power-saving mode (see Section 2.2). When the thread is in a power-saving level that allows the state of the LPCR to be lost, it is implementation-specific whether the following events, when enabled, cause exit, or whether only a system-reset exception causes exit.

- **External**
Decrementer
Directed Privileged Doorbell
Directed Hypervisor Doorbell
Directed Ultravisor Doorbell
Hypervisor Maintenance
Hypervisor Virtualization exception
Implementation-specific

SRR1 indicates the exception that caused exit from power-saving mode as specified below.

The following registers are set:

**SRR0**
If the interrupt did not occur when the thread was in power-saving mode, set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present; if the interrupt occurred when the thread was in a power-saving mode that was entered with PSSCR bit ESL=0, and fields RL, MTL, and PSLL set to values that do not allow state loss, set to the effective address of the instruction following the stop instruction; otherwise, set to an undefined value.

If the interrupt occurred while the thread was in power-saving mode, set to the effective address of the instruction following the **stop** instruction when **stop** is executed with PSSCR bit ESL=0 and fields RL, MTL, and PSLL set to values that do not allow state loss; otherwise, set to an undefined value.

--- Programming Note ---

Whenever **stop** is executed in privileged non-hypervisor state, the hypervisor typically sets both PSSCRESL and PSSCREC to 0, and sets RL and MTL to values that do not cause state loss. If an interrupt causes exit to power-saving mode (either because the interrupt was a System Reset or Machine Check interrupt or MSREE=1), then SRR0 for that interrupt contains the effective address of the instruction immediately following **stop**.

**SRR1**

**33** Implementation-dependent.

**34:36** Set to 0.

**42:45** If the interrupt did not occur when the thread was in power-saving mode, set to an implementation-specific value. If the interrupt occurred when the thread was in power-saving mode, set to indicate the exception that caused exit from power-saving mode as shown below:

<table>
<thead>
<tr>
<th>SRR1 42:45</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Reserved</td>
</tr>
<tr>
<td>0001</td>
<td>Directed Ultravisor Doorbell</td>
</tr>
<tr>
<td>0010</td>
<td>Implementation specific</td>
</tr>
<tr>
<td>0011</td>
<td>Directed Hypervisor Doorbell</td>
</tr>
<tr>
<td>0100</td>
<td>System Reset</td>
</tr>
<tr>
<td>0101</td>
<td>Directed Privileged Doorbell</td>
</tr>
<tr>
<td>0110</td>
<td>Decrementer</td>
</tr>
<tr>
<td>0111</td>
<td>Reserved</td>
</tr>
<tr>
<td>1000</td>
<td>External</td>
</tr>
<tr>
<td>1001</td>
<td>Hypervisor Virtualization</td>
</tr>
<tr>
<td>1010</td>
<td>Hypervisor Maintenance</td>
</tr>
<tr>
<td>1011</td>
<td>Reserved</td>
</tr>
<tr>
<td>1100</td>
<td>Implementation specific</td>
</tr>
<tr>
<td>1101</td>
<td>Reserved</td>
</tr>
<tr>
<td>1110</td>
<td>Implementation specific</td>
</tr>
<tr>
<td>1111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**46:47** Set to indicate whether the interrupt occurred when the thread was in power-saving mode and, if so, the extent to which resource state was maintained while the thread was in power-saving mode, as follows:

| 00 | The interrupt did not occur when the thread was in power-saving mode. |
| 01 | The interrupt occurred when the thread was in power-saving mode. The state of all resources was maintained as if the thread was not in power-saving mode. |
The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, but the state of all hypervisor resources, including the DEC, HDEC, TB, PURR, SPURR, and VTB, was maintained as if the thread was not in power-saving mode and the state of all other resources is such that the hypervisor can resume execution. (See Section 2.6 for the list of hypervisor resources.)

The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, and the state of some hypervisor resources was not maintained or the state of some resources is such that the hypervisor cannot resume execution.

Programming Note

Although the resources that are maintained in power-saving levels that allow loss of state are implementation-dependent, the hypervisor can avoid implementation-dependence in the portion of the System Reset and Machine Check interrupt handlers that recover from having been in power-saving mode by using the contents of SRR1_{46:47}, to determine what state to restore. To avoid implementation-dependence, the hypervisor must assume that only the resources indicated in SRR1_{46:47} have been preserved.

The interrupt occurred when the thread was in power-saving mode. If the interrupt did not occur while the thread was in a power-saving level that was entered when PSSCR_{EC}=1, loaded from bit 62 of the MSR if the thread is in a recoverable state; otherwise set to 0. If the interrupt occurred while the thread was in a power-saving level that was entered when PSSCR_{EC}=1, set to 1 if the thread is in a recoverable state; otherwise set to 0.

Others

Loaded from the MSR.

MSR

See Figure 67 on page 1261.

In addition, if the interrupt occurs when the thread is in a power-saving level that was entered when PSSCR_{EC}=1 and is caused by an exception other than a System Reset exception, all other registers, except HSRR0 and HSRR1, that would be set by the corresponding interrupt if the exception occurred when the thread was not in power-saving mode are set by the System Reset interrupt, and are set to the values to which they would be set if the exception occurred when the thread was not in power-saving mode.

Execution resumes at effective address 0x0000_0000_0000_0100.

The means for software to distinguish between power-on Reset and other types of System Reset are implementation-dependent.

7.5.2 Machine Check Interrupt

The causes of Machine Check interrupts are implementation-dependent. For example, a Machine Check interrupt may be caused by a reference to a storage location that contains an uncorrectable error or does not exist (see Section 6.6), or by an error in the storage subsystem. In some cases, processor designers may choose to present what would normally be considered a storage exception, and reported as an [H]DSI or [H]ISI, as a Machine Check interrupt instead, often because of the perceived severity of the programming error or the difficulty of the verification and testing associated with reporting the error as an [H]DSI or [H]ISI. If the decision to report the exception as a Machine Check interrupt is made when the scenario is first added to the architecture, the exception may never be documented in a storage interrupt description. One such case is an attempt to access storage mapped to an accelerator as anything other than an operand of copy or paste.

When the thread is not in power-saving mode, Machine Check interrupts are enabled when MSR_{ME}=1; if MSR_{ME}=0 and a Machine Check exception occurs, the thread enters the Checkstop state. When the thread is in a power-saving level that does not allow loss of hypervisor state, Machine Check interrupts are treated as enabled when LPCR_{S1}=1 and cannot occur when LPCR_{S1}=0. When the thread is in a power-saving level that allows loss of hypervisor state, it is implementation-specific whether Machine Check interrupts are treated as enabled LPCR_{S1}=1 or if they cannot occur. If a Machine Check exception occurs while the thread is in power-saving mode and the Machine Check exception is not enabled to cause exit from power-saving mode, the result is implementation specific.

The Checkstop state may also be entered if an access is attempted to a storage location that does not exist (see Section 6.6), or if an implementation-dependent hardware error occurs that prevents continued operation.

Disabled Machine Check (Checkstop State)

When a thread is in Checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the thread. Some implementations may preserve some or all of the internal state of
the thread when entering Checkstop state, so that the state can be analyzed as an aid in problem determination.

**Enabled Machine Check**

If a Machine Check exception causes an interrupt that is not context synchronizing or causes the loss of a Direct External exception, or if the state of the thread has been corrupted, the interrupt is not recoverable.

The following registers are set:

**SRR0**

- If the interrupt occurred when the thread was not in a power-saving mode, or was in a power-saving mode that was entered with PSSCR bit ESL=0, and fields RL, MTL, and PSLL set to values that do not allow state loss, set on a "best effort" basis to the effective address of some instruction that was executing or was about to be executed when the Machine Check exception occurred; otherwise set to an undefined value.

**SRR1**

- If SRR0 is set to the effective address of an instruction for which the instruction length can easily be reported, set to 0 if the instruction is a word instruction and to 1 if the instruction is a prefixed instruction; otherwise set to an undefined value.

**46:47**

- Set to indicate whether the interrupt occurred when the thread was in power-saving mode and, if so, the extent to which resource state was maintained while the thread was in power-saving mode, as follows.

  - **00** The interrupt did not occur when the thread was in power-saving mode.
  - **01** The interrupt occurred when the thread was in power-saving mode. The state of all resources was maintained as if the thread was not in power-saving mode.
  - **10** The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, but the state of all hypervisor resources, including the DEC, HDEC, TB, PURR, SPURR, and VTB, was maintained as if the thread was not in power-saving mode and the state of all other resources is such that the hypervisor can resume execution. (See Section 2.6 for the list of hypervisor resources.)
  - **11** The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, and the state of some hypervisor resources was not maintained or the state of some resources is such that the hypervisor cannot resume execution.

**Programming Note**

Although the resources that are maintained in power-saving mode (except when all resources are maintained) are implementation-dependent, the hypervisor can avoid implementation-dependence in the portion of the System Reset and Machine Check interrupt handlers that recover from having been in power-saving mode by using the contents of SRR146:47 to determine what state to restore. (To avoid implementation-dependence in the portion of the hypervisor that enters power-saving mode, the hypervisor must use the specification of the four instructions to determine what state to save.)

**Others**

- Set to an implementation-dependent value.

**MSR**

- See Figure 67.

**DSISR**

- Set to an implementation-dependent value.

**DAR**

- Set to an implementation-dependent value.

**ASDR**

- Set to an implementation-dependent value.
Execution resumes at effective address 0x0000_0000_0000_0200.

A Machine Check interrupt caused by the existence of multiple SLB entries or TLB entries (or similar entries in implementation-specific translation caches) which translate a given effective or virtual address (see Sections 6.7.8.2 and 6.7.9.2.) must occur while still in the context of the partition that caused it. The interrupt must be presented in a way that permits continuing execution, with damage limited to the causing partition. Treating the exception as instruction-caused will achieve these requirements.

---

**Programming Note**

If a Machine Check interrupt is caused by an error in the storage subsystem, the storage subsystem may return incorrect data, which may be placed into registers. This corruption of register contents may occur even if the interrupt is recoverable.

---

### 7.5.3 Data Storage Interrupt (DSI)

A Data Storage interrupt occurs when no higher priority exception exists and either

(a) (MSRHV PR=0b10) & (MSRDR=0)) and the data access cannot be performed, or

(b) HPT translation is being performed, the value of the expression

\[
(\text{MSR}_{HV} \text{ PR}=0b10) \land (\neg \text{VPM}) \land \text{PRTE}_{V} \land \text{MSR}_{DR}=1
\]

is 1, and a data access cannot be performed, except for the case of MSR$_{HV}$PR=0b10, VPM=0, LPCR$_{KBV}=1$, and a Virtual Page Class Key Storage Protection exception exists or

(c) Radix Tree translation is being performed, and either a Data Address Watchpoint match occurs, an attempt is made to execute an AMO with an invalid function code, an attempt is made to perform a copy-paste transfer other than from main storage to a properly initiated accelerator, a problem other than page fault or unsupported MMU configuration caused by (RTS or RPDS in) the PATE occurs attempting to access the LPID=0 process table, or process-scoped translation prevents the data access from being performed

for any of the following reasons that can occur in the respective translation state except for a PTEG access causing a Secure Memory Protection exception when VPM=0:

- Data address translation is enabled (MSR$_{DR}=1$) and the effective or virtual address of any byte of the storage location specified by a Load, Store, icbi, dbz, dcbs, or dcdf instruction cannot be translated to a real address because no valid PTE was found for the process-scoped Radix Tree translation or HPT translation with VPM off.
- The address of the appropriate process table entry or segment table entry group cannot be translated when HR=0 and either VPM=0 or the process table entry is invalid (independent of VPM).
- The effective address specified by a lq, lq, lwat, lbarx, lbarx, lbarx, lwat, lbart, sbcx, stbcx, stbcx, stwcx, stbcx, or stqcx instruction refers to storage that is Write Through Required or Caching Inhibited; or the effective address specified by a copy or paste instruction refers to storage that is Caching Inhibited; or the effective address specified by a lwat, ldat, stwat, or stdat instruction refers to storage that is Guarded.
- An accelerator is specified as the source of a copy instruction or an attempt is made to access an accelerator that is not properly configured for the software’s use.
- The access violates Basic Storage Protection.
- The access violates Virtual Page Class Key Storage Protection and LPCR$_{KBV}=0$.
- The access violates Radix Tree Storage Protection because the process-scoped PTE does not permit the access.
- The access violates Secure Memory Protection.
- The process- and partition-scoped page attributes conflict.
- An unsupported radix tree configuration is found for the process-scoped tables, or, if effLPID=0, for the partition-scoped tables. (Note that this condition may not be detected until the associated values are about to cause a functional problem for the processor.)
- A reference or change bit update cannot be performed in a process-scoped PTE.
- A Data Address Watchpoint match occurs.
- An attempt is made to execute a Load Atomic or Store Atomic instruction with an invalid function code.
- An attempt is made to execute a Fixed-Point Load or Store Caching Inhibited instruction with MSR$_{DR}=1$ or specifying a storage location that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded.

A Data Storage interrupt also occurs when no higher priority exception exists and an attempt is made to execute a Load Atomic or Store Atomic instruction specifying an invalid function code.

---

**Programming Note**

When an attempt to execute a Load Atomic or Store Atomic instruction containing an invalid function code (see Figures 3 and 4 in Book II) causes a DSI, the condition is very similar to an invalid form of an instruction. As a result, this instance of DSI occurs with a high priority that blocks the translation process and prevents Reference and Change bit updates.
If a *stbcx*, *sthcx*, *stwcx*, or *stdcx*, would not perform its store in the absence of a Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional Store to the specified effective address would cause a Data Storage interrupt, it is implementation-dependent whether a Data Storage interrupt occurs.

If the XER specifies a length of zero for an indexed Move Assist instruction, a Data Storage interrupt does not occur.

The following registers are set:

- **SRR0**: Set to the effective address of the instruction that caused the interrupt.
- **SRR1**:  
  - 33: Set to 0.  
  - 34: Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.  
  - 35:36: Set to 0.  
  - 42:47: Set to 0.  
  - Others: Loaded from the MSR.
- **MSR**: See Figure 67.
- **DSISR**:  
  - 32: Set to 0.  
  - 33: Set to 1 if MSRDR=1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0.  
  - 34: Set to 1 if the process- and partition-scoped page attributes conflict; otherwise set to 0.  
  - 35: Set to 0.  
  - 36: Set to 1 if the access is not permitted by Figure 45 or the privilege, read, or read/write bits in Figure 46 as appropriate; otherwise set to 0.  
  - 37: Set to 1 if the access is due to a *lq*, *stq*, *lqat*, *ldat*, *lbarx*, *lqax*, *stdat*, *stdcx*, *stdat*, or *stqcx*. instruction that addresses storage that is Write Through Required or Caching Inhibited; or if the access is due to a *copy* or *paste*. instruction that addresses storage that is Caching Inhibited; or if the access is due to a *lqat*, *ldat*, *stqat*, *stqcx*, *stdat*, or *stdat*. instruction that addresses storage that is Guarded; otherwise set to 0.  
  - 38: Set to 1 for a *Store*, *dcbz*, or *Load/Store Atomic* instruction; otherwise set to 0.  
  - 39:40: Set to 0.  
  - 41: Set to 1 if a Data Address Watchpoint match occurs; otherwise set to 0.  
  - 42: Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.  
  - 43: Set to 1 if the access is not permitted by Secure Memory Protection; otherwise set to 0.  
  - 44: Set to 1 if an unsupported radix tree configuration is found during the translation process; otherwise set to 0.  
  - 45: Set to 1 if an attempt to atomically set a reference or change bit fails; otherwise set to 0.

### Programming Note

The number of attempts hardware makes to atomically set reference and change bits before triggering this exception is implementation dependent. The POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 6.7.12.

- **46**: Set to 1 if the address of the appropriate process table entry or segment table entry group cannot be translated when VPM=0 and HR=0, or the process table entry is invalid (independent of VPM) when HR=0.
- **47:59**: Set to 0.
- **60**: Set to 1 if an accelerator is specified as the source of a *copy* instruction or an attempt is made to access an accelerator that is not properly configured for the software’s use; otherwise set to 0. These exceptions are presented differently from most instruction-caused exceptions. See Section 4.4, “Copy-Paste Facility”, in Book II for details. Additional information may be retained by the platform if the accelerator is not properly configured.
- **61**: Set to 1 if an attempt is made to execute a *Load Atomic* or *Store Atomic* instruction specifying an invalid function code; otherwise set to 0.
- **62**: Set to 1 if an attempt is made to execute a *Fixed-Point Load* or *Store Caching Inhibited* instruction with MSRDR=1 or specifying a storage location that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded.
- **63**: Set to 0.
- **64**: Set to the effective address of a storage element as described in the following list. The list should be read from the top down; the DAR is set as described by the first item that corresponds to an exception that is reported in the DSISR. For example, if a *Load Word* instruction causes a storage protection violation and a Data Address Watchpoint match (and both are reported in the DSISR), the DAR is set to the effective address of the instruction that caused the interrupt.

---

**Chapter 7. Interrupts**  1267

[Revision Information]
address of a byte in the first aligned double-word for which access was attempted in the page that caused the exception.
- undefined, for Load Atomic or Store Atomic instruction specifying an invalid function code
- undefined, when DSISRb0=1
- a Data Storage exception occurs for reasons other than a Data Address Watchpoint match
  - a byte in the block that caused the exception, for a Cache Management instruction
  - a byte in the first aligned double-word for which access was attempted in the page that caused the exception, for a Load or Store instruction ("first" refers to address order: see Section 7.7).
- the first byte of overlap between the operand and the matching watched range, for a Data Address Watchpoint match

For the cases in which the DAR is specified above to be set to a defined value, if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

If multiple Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the DSISR. However, if one or more Data Storage exceptions occur together with a Virtualized Page Class Key Storage Protection exception that occurs when LPCRKBV=1 and Virtualized Partition Memory is disabled by VPM=0, an HDSI results, and all of the exceptions are reported in the HDSISR.

Execution resumes at effective address 0x0000_0000_0000_0300, possibly offset as specified in Figure 68.

### 7.5.4 Data Segment Interrupt

For Paravirtualized HPT Translation, a Data Segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because data address translation is enabled and the effective address of any byte of the storage location specified by a Load, Store, icbi, dcbz, dcbs, or dcbf instruction cannot be translated to a virtual address because the SLB search fails and, if UPRT=1, the Segment Table search fails after the STEG has been accessed.

For Radix Tree Translation, a Data Segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because for the effective address specified by a Load, Store, icbi, dcbz, dcbs, or dcbf instruction, data address translation is enabled and either EA0:1=0b01 or 0b10 when MSRHV PR ≠ 0b10 or EA0:1=0b00 when MSRHV PR=0b10 and LPIDR≠0, or EA2:63 is outside the range translated by the appropriate Radix Tree.

If a stb.cx, stbcx, stwcx, stdcx, or stqcx. would not perform its store in the absence of a Data Segment interrupt and a non-conditional Store to the specified effective address would cause a Data Segment interrupt, it is implementation-dependent whether a Data Segment interrupt occurs.

If the XER specifies a length of zero for an indexed Move Assist instruction, a Data Segment interrupt does not occur.

The following registers are set:

| SRR0 | Set to the effective address of the instruction that caused the interrupt. |
| SRR1 | 33  | Set to 0. |
|      | 34  | Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction. |
|      | 35:36 | Set to 0. |
|      | 42:47 | Set to 0. |
| Others | Loaded from the MSR. |

| MSR | See Figure 67. |
| DAR | Set to undefined value. |

The following registers are set:

| SRR0 | Set to the effective address of a storage element as described in the following list. |
| SRR1 | 33  | Set to 0. |
|      | 34  | Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction. |
|      | 35:36 | Set to 0. |
|      | 42:47 | Set to 0. |
| Others | Loaded from the MSR. |

A Data Segment interrupt occurs if MSRDR=1 and the translation of the effective address of any byte of the specified storage location is not found in the SLB (or in any implementation-specific address translation lookaside information).

---

**Programming Note**

A Data Segment interrupt occurs if MSRDR=1 and the translation of the effective address of any byte of the specified storage location is not found in the SLB (or in any implementation-specific address translation lookaside information).
7.5.5 Instruction Storage Interrupt (ISI)

An Instruction Storage interrupt occurs when no higher priority exception exists and either
(a) \((\text{MSR}_{\text{HVPR}}=0b10) \& (\text{MSR}_{\text{IR}}=0))\) and the next instruction to be executed cannot be fetched, or
(b) HPT Translation is being performed, the value of the expression
\(((\text{MSR}_{\text{HVPR}}=0b10) \lor ((\neg \text{VPM}) \lor \neg \text{PRTEV}) \& \text{MSR}_{\text{IR}}))\) is 1, and the next instruction to be executed cannot be fetched, or
(c) Radix Tree translation is being performed and either a problem other than page fault or unsupported MMU configuration caused by (RTS or RPDS in) the PATE occurs attempting to access the LPID=0 process table or process-scoped translation prevents the next instruction to be executed from being fetched

for any of the following reasons that can occur in the respective translation state except for a PTEG access causing a Secure Memory Protection exception when VPM=0.

- Instruction address translation is enabled and the effective or virtual address cannot be translated to a real address because no valid PTE was found for the process-scoped Radix Tree translation or HPT translation with VPM off.
- The address of the appropriate process table entry or segment table entry group cannot be translated when HR=0 and either VPM=0 or the process table entry is invalid (independent of VPM).
- The access violates Basic Storage Protection.
- The access violates Virtual Page Class Key Storage Protection.
- The access violates Radix Tree Translation Storage Protection.
- The access violates Secure Memory Protection.
- The process- and partition-scoped page attributes conflict.
- An unsupported radix tree configuration is found for the process-scoped tables, or, if effLPID=0, for the partition-scoped tables. (Note that this condition may not be detected until the associated values are about to cause a functional problem for the processor.)
- A reference bit update cannot be performed in a process-scoped PTE.
- The instruction is a prefixed instruction and is in storage that is Caching Inhibited.

The following registers are set:

<table>
<thead>
<tr>
<th>SRR0</th>
<th>Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR1</td>
<td>Set to 1 if MSR_{IR}=1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0.</td>
</tr>
<tr>
<td>33</td>
<td>Set to 1 if the process- and partition-scoped page attributes conflict; otherwise set to 0.</td>
</tr>
<tr>
<td>34</td>
<td>Set to 1 if the access is to No-execute (as indicated by the N bit in the segment table entry or the N bit in the HPT PTE or the Execute and Privilege bits in the EAA field of the Radix PTE and IAMR key 0) or Guarded storage, or is to Caching Inhibited storage and is for a prefixed instruction; otherwise set to 0.</td>
</tr>
<tr>
<td>35</td>
<td>Set to 1 if the access is not permitted by Figure 45 or the privilege or execute bits in Figure 46 as appropriate; otherwise set to 0.</td>
</tr>
<tr>
<td>36</td>
<td>Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.</td>
</tr>
<tr>
<td>42</td>
<td>Set to 1 if the access is not permitted by Secure Memory Protection; otherwise set to 0.</td>
</tr>
<tr>
<td>43</td>
<td>Set to 1 if an unsupported radix tree configuration is found during the translation process; otherwise set to 0.</td>
</tr>
<tr>
<td>44</td>
<td>Set to 1 if an attempt to atomically set a reference bit fails; otherwise set to 0.</td>
</tr>
<tr>
<td>45</td>
<td>Set to 1 if the address of the appropriate process table entry or segment table entry group cannot be translated when VPM=0 and HR=0, or the process table entry is invalid (independent of VPM) when HR=0.</td>
</tr>
<tr>
<td>46</td>
<td>Set to 1 if the address of the appropriate process table entry or segment table entry group cannot be translated when VPM=0 and HR=0, or the process table entry is invalid (independent of VPM) when HR=0.</td>
</tr>
<tr>
<td>47</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>Others</td>
<td>Loaded from the MSR.</td>
</tr>
<tr>
<td>MSR</td>
<td>See Figure 67.</td>
</tr>
</tbody>
</table>

Programming Note

The number of attempts hardware makes to atomically set reference and change bits before triggering this exception is implementation dependent. The POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 6.7.12.

46 Set to 1 if the address of the appropriate process table entry or segment table entry group cannot be translated when VPM=0 and HR=0, or the process table entry is invalid (independent of VPM) when HR=0.

47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 67.

If multiple Instruction Storage exceptions occur due to attempting to fetch a single instruction, any one or more of the bits corresponding to these exceptions may be set to 1 in SRR1.

Execution resumes at effective address 0x0000_0000_0000_0400, possibly offset as specified in Figure 68.
7.5.6 Instruction Segment Interrupt

For Paravirtualized HPT Translation, an Instruction Segment interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched because instruction address translation is enabled and the effective address cannot be translated to a virtual address because the SLB search fails and, if UPRT=1, the Segment Table search fails after the STEG has been accessed.

For Radix Tree Translation, an Instruction Segment interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched because instruction address translation is enabled and either $EA_{0:1}=0b01$ or $0b10$ when $MSR_{HVPR} \neq 0b10$ or $EA_{0:1}=0b00$ when $MSR_{HVPR}=0b10$ and $LPIDR=0$, or $EA_{2:63}$ is outside the range translated by the appropriate Radix Tree.

The following registers are set:

**SRR0** Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).

**SRR1**
- $33:36$ Set to 0.
- $42:47$ Set to 0.
- Others Loaded from the MSR.

**MSR** See Figure 67 on page 1261.

Execution resumes at effective address $0x0000_0000_0000_0480$, possibly offset as specified in Figure 68.

---

**Programming Note**

An Instruction Segment interrupt occurs if $MSR_{SR}=1$ and the translation of the effective address of the next instruction to be executed is not found in the SLB (or in any implementation-specific address translation lookaside information).

7.5.7 External Interrupt

An External interrupt is classified as being either a Direct External interrupt or a Mediated External interrupt. Throughout this Book, usage of the phrase 'External interrupt', without further classification, refers to both a Direct External interrupt and a Mediated External interrupt.

7.5.7.1 Direct External Interrupt

A Direct External interrupt occurs when no higher priority exception exists, a Direct External exception exists, and the value of the expression

$$MSR_{EE} \land \neg (MSR_{HV} \land \neg MSR_{PR} \land LPCR_{HEIC}) \lor \neg (LPES) \lor (\neg MSR_{HV} \lor MSR_{PR})$$

is one. The occurrence of the interrupt does not cause the exception to cease to exist.

---

**Programming Note**

When HEIC=1, Direct External exceptions will not result in external interrupts when the processor is in hypervisor state even if $MSR_{EE}=1$. This enables the Hypervisor Virtualization interrupt handler to prevent External interrupts from occurring during the Hypervisor Virtualization interrupt handler.

When LPES=0, the following registers are set:

**HSRR0** Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

**HSRR1**
- $33:36$ Set to 0.
- $42:47$ Set to 0.
- Others Loaded from the MSR.

**MSR** See Figure 67 on page 1261.

When LPES=1, the following registers are set:

**SRR0** Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

**SRR1**
- $33:36$ Set to 0.
- $42:47$ Set to 0.
- Others Loaded from the MSR.

**MSR** See Figure 67 on page 1261.

Execution resumes at effective address $0x0000_0000_0000_0500$, possibly offset as specified in Figure 68.

---

**Programming Note**

Because the value of $MSR_{EE}$ is always 1 when the thread is in problem state, the simpler expression

$$MSR_{EE} \land \neg (MSR_{HV} \land \neg MSR_{PR} \land LPCR_{HEIC}) \lor \neg (LPES) \lor (\neg MSR_{HV} \lor MSR_{PR})$$

is equivalent to the expression given above.
Programming Note

The Direct External exception has the same meaning as the External exception in versions of the architecture prior to Version 2.05.

7.5.7.2 Mediated External Interrupt

A Mediated External interrupt occurs when no higher priority exception exists, a Mediated External exception exists (see the definition of LPCRMER in Section 2.2), and the value of the expression

\[ \text{MSR}_{EE} \& (\neg(\text{MSR}_{HV}) \mid \text{MSR}_{PR}) \]

is one. The occurrence of the interrupt does not cause the exception to cease to exist.

When LPES=0, the following registers are set:

- **HSRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **HSRR1**:
  - 0x33:36 Set to 0.
  - 0x42 Set to 1.
  - 0x43:47 Set to 0.
- **Others**: Set to 0.
- **MSR**: See Figure 67 on page 1261.

When LPES=1, the following registers are set:

- **SRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **SRR1**: 
  - 0x33:36 Set to 0.
  - 0x42 Set to 1.
  - 0x43:47 Set to 0.
- **Others**: Loaded from the MSR.
- **MSR**: See Figure 67 on page 1261.

Execution resumes at effective address 0x0000_0000_0000_0500, possibly offset as specified in Figure 68.

7.5.8 Alignment Interrupt

Most causes of Alignment interrupt involve storage operands, and many of those causes involve the alignment thereof. Storage operand alignment is defined in Section 1.10.1 of Book I. Another cause of Alignment interrupt is attempt to execute a prefixed instruction that crosses a 64-byte address boundary.

An Alignment interrupt occurs when no higher priority exception exists and an attempt is made to execute an instruction in a manner that is required, by the instruction description, to cause an Alignment interrupt, or to execute a prefixed instruction that crosses a 64-byte boundary. These cases are as follows.

- A Load/Store Multiple instruction that is executed in Little-Endian mode
- A Move Assist instruction that is executed in Little-Endian mode, unless the string length is zero
- A copy, paste, iwat, ldat, lhax, lwax, ldarx, lqarx, stwat, stdat, sthcx., stwcx., or stqcx. instruction that has an unaligned storage operand, unless execution of the instruction yields boundedly undefined results
- The operand(s) of a Load Atomic or Store Atomic instruction cross(es) a 32-byte boundary.
- A prefixed instruction that is at an effective address equal to 60 modulo 64.

An Alignment interrupt may occur when no higher priority exception exists and a data access cannot be performed for any of the following reasons.

- The storage operand of lfdp, lfdpx, stfdp, stfdpx, lxsihzx, or stxsihx is unaligned.
- The storage operand of lq or stq is unaligned.
- The storage operand of a Floating-Point Storage Access or VSX Storage Access instruction other than lfdp, lfdpx, stfdp, stfdpx, lxsihzx, lxsibzx, stxsihx, or stxsbx is not word-aligned.
- The storage operand of a Load/Store Multiple Word instruction is not word-aligned and the thread is in Big-Endian mode.
- The storage operand of a Load/Store Multiple Doubleword instruction is not doubleword-aligned and the thread is in Big-Endian mode.
- The storage operand of a Load/Store Multiple, lfdp, lfdpx, lxvi, lxvl, stfdp, stfdpx, stxvi, stxvl, or dcbz instruction is in storage that is Write Through Required or Caching Inhibited.
- The storage operand of a Move Assist instruction is in storage that is Write Through Required or Caching Inhibited and has length greater than zero.
- The storage operand of a Load or Store instruction is unaligned and is in storage that is Write Through Required or Caching Inhibited.
- The storage operand of a Storage Access instruction crosses a segment boundary, or crosses a boundary between virtual pages that have different storage control attributes.

The following registers are set:

- **SRR0**: Set to the effective address of the instruction that caused the interrupt.
- **SRR1**: 
  - 0x33 Set to 0.
  - 0x34 Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.
  - 0x35 Set to 1 if the instruction that caused the interrupt is a prefixed instruction that is at
an effective address equal to 60 modulo 64; otherwise set to 0.

36
Set to 0.

42:47
Set to 0.

Others
Loaded from the MSR.

MSR
See Figure 67.

DAR
Set to the effective address computed by the instruction, unless the instruction is a prefixed instruction at an effective address equal to 60 modulo 64, in which case set to an undefined value except as described in the next sentence. If the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

Execution resumes at effective address 0x0000_0000_0000_0600, possibly offset as specified in Figure 68.

---

**Programming Note**

If an Alignment interrupt occurs for a case in the second bulleted list above, the Alignment interrupt handler should emulate the instruction. The emulation must satisfy the atomicity requirements described in Section 1.4 of Book II.

If an Alignment interrupt occurs for a case in the first bulleted list above, the Alignment interrupt handler must not attempt to emulate the instruction, but instead should treat the instruction as a programming error.

---

**7.5.9 Program Interrupt**

A Program interrupt occurs when no higher priority exception exists and one of the following exceptions arises during execution of an instruction:

**Floating-Point Enabled Exception**

A Floating-Point Enabled Exception type Program interrupt is generated when the value of the expression

\[ (\text{MSRFE}0) \land (\text{MSRFE}1) \land \text{FPSCR}_{\text{FEX}} \]

is 1. FPSCR_{FEX} is set to 1 by the execution of a floating-point instruction that causes an enabled exception, including the case of a Move To FPSCR instruction that causes an exception bit and the corresponding enable bit both to be 1.

**Privileged Instruction**

The following applies if the instruction is executed when MSR_{PR} = 1.

A Privileged Instruction type Program interrupt is generated when execution is attempted of a privileged instruction, or of an mtsp or mfspr instruction that specifies an SPR that is privileged for the operation or that specifies PTCR, DAWRn, DAWRXn, or CIABR when those SPRs are ultravisor privileged for the operation.

The following applies if the instruction is executed when MSR_{HV PR} = 0b00 or when MSR_{S HV PR} = 0b010.

A Privileged Instruction type Program interrupt is generated when execution is attempted of an ultravisor privileged instruction, or of an mtsp or mfspr instruction that specifies an SPR, other than PTCR, DAWRn, DAWRXn, and CIABR, that is ultravisor privileged for the operation.

---

**Programming Note**

These are the only cases in which a Privileged Instruction type Program interrupt can be generated when MSR_{PR}=0. They can be distinguished from other causes of Privileged Instruction type Program interrupts by examining SRR1_{49} (the bit in which MSR_{PR} was saved by the interrupt).

---

**Trap**

A Trap type Program interrupt is generated when any of the conditions specified in a Trap instruction is met.

The following registers are set:

**SRR0**

For all Program interrupts except a Floating-Point Enabled Exception type Program interrupt, set to the effective address of the instruction that caused the corresponding exception.

For a Floating-Point Enabled Exception type Program interrupt, set as described in the following list.

- If MSR_{FE0 FE1} = 0b00, FPSCR_{FEX} = 1, and an instruction is executed that changes MSR_{FE0 FE1} to a nonzero value, set to the effective address of the instruction that the thread would have attempted
to execute next if no interrupt conditions were present.

**Programming Note**
Recall that all instructions that can alter MSR\textsubscript{FE0 FE} are context synchronizing, and therefore are not initiated until all preceding instructions have reported all exceptions they will cause.

- If MSR\textsubscript{FE0 FE} = 0b11, set to the effective address of the instruction that caused the Floating-Point Enabled Exception.
- If MSR\textsubscript{FE0 FE} = 0b01 or 0b10, set to the effective address of the first instruction that caused a Floating-Point Enabled Exception since the most recent time FFSR\textsubscript{FE} was changed from 1 to 0 or of some subsequent instruction.

**Programming Note**
If SRR0 is set to the effective address of a subsequent instruction, that instruction will not be beyond the first such instruction at which synchronization of floating-point instructions occurs. (Recall that such synchronization is caused by Floating-Point Status and Control Register instructions, as well as by execution synchronizing instructions and events.)

<table>
<thead>
<tr>
<th>SRR1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>34</td>
<td>Except for the cases in which SRR0 is not necessarily set to the effective address of the instruction that caused the exception (first and third bullets in the description of how SRR0 is set), set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction; for the cases in which SRR0 is not necessarily set to the effective address of the instruction that caused the exception, set to an undefined value.</td>
</tr>
<tr>
<td>35:36</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>42</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>43</td>
<td>Set to 1 for a Floating-Point Enabled Exception type Program interrupt; otherwise set to 0.</td>
</tr>
<tr>
<td>44</td>
<td>Set to 0.</td>
</tr>
</tbody>
</table>

**Others**
Loaded from the MSR.

Exactly one of bits 43, 45, and 46 is set to 1.

**MSR**
See Figure 67 on page 1261.

Execution resumes at effective address 0x0000_0000_0000_0700, possibly offset as specified in Figure 68.
7.5.10 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point loads, stores, and moves), and MSR_FFP=0.

The following registers are set:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction that caused the interrupt.</td>
</tr>
<tr>
<td>SRR1</td>
<td>Set to 0.</td>
</tr>
<tr>
<td></td>
<td>Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.</td>
</tr>
<tr>
<td>Others</td>
<td>Loaded from the MSR.</td>
</tr>
<tr>
<td>MSR</td>
<td>See Figure 67 on page 1261.</td>
</tr>
</tbody>
</table>

Execution resumes at effective address 0x0000_0000_0000_0800, possibly offset as specified in Figure 68.

Programming Note

In versions of the architecture that precede V. 2.05, the conditions that now cause a Hypervisor Emulation Assistance interrupt with HSRR145=0 instead caused an "Illegal Instruction type Program interrupt". This was a Program interrupt for which registers (SRR0, SRR1, and the MSR) were set as described above for the Privileged Instruction type Program interrupt, except that SRR144 was set to 1 and SRR145 was set to 0. Thus older operating systems have code to handle these conditions, at the Program interrupt vector location. For this reason, if a Hypervisor Emulation Assistance interrupt occurs with HSRR145=0 when the thread is not in hypervisor state, for an instruction that the hypervisor determines should be handled by the operating system, the hypervisor is expected to pass control to the operating system at the operating system’s Program interrupt vector location, with all registers (SRR0, SRR1, MSR, GPRs, etc.) set as if the instruction had caused a Privileged Instruction type Program interrupt, except with SRR144:45 set to 0b10. (The Hypervisor Emulation Assistance interrupt was added to the architecture in V. 2.05, and the Illegal Instruction type Program interrupt was removed from the architecture in V. 2.06. In V. 2.05 the Hypervisor Emulation Assistance interrupt was optional: implementations that supported it generated it as described in V. 2.06, and never generated an Illegal Instruction type Program interrupt; implementations that did not support it generated an Illegal Instruction type Program interrupt as described above.)
7.5.11 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists, a Decrementer exception exists, and MSR$_{EE}$=1.

The following registers are set:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.</td>
</tr>
<tr>
<td>SRR1</td>
<td>33:36 Set to 0. 42:47 Set to 0.</td>
</tr>
<tr>
<td>Others</td>
<td>Loaded from the MSR.</td>
</tr>
</tbody>
</table>

MSR

See Figure 67 on page 1261.

Execution resumes at effective address 0x0000_0000_0000_0900, possibly offset as specified in Figure 68.

7.5.12 Hypervisor Decrementer Interrupt

A Hypervisor Decrementer interrupt occurs when no higher priority exception exists, a Hypervisor Decrementer exception exists, and the value of the following expression is 1.

$$(\text{MSR}_E \lor \neg (\text{MSR}_H) \lor \text{MSR}_P) \& \text{HDICE}$$

The following registers are set:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSRR0</td>
<td>Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.</td>
</tr>
<tr>
<td>HSRR1</td>
<td>33:36 Set to 0. 42:47 Set to 0.</td>
</tr>
<tr>
<td>Others</td>
<td>Loaded from the MSR.</td>
</tr>
</tbody>
</table>

MSR

See Figure 67 on page 1261.

Execution resumes at effective address 0x0000_0000_0000_0980, possibly offset as specified in Figure 68.

7.5.13 Directed Privileged Doorbell Interrupt

A Directed Privileged Doorbell interrupt occurs when no higher priority exception exists, a Directed Privileged Doorbell exception is present, and MSR$_{EE}$=1. Directed Privileged Doorbell exceptions are generated when Directed Privileged Doorbell messages (see Chapter 11) are received and accepted by the thread.

The following registers are set:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.</td>
</tr>
<tr>
<td>SRR1</td>
<td>33:36 Set to 0. 42:47 Set to 0.</td>
</tr>
<tr>
<td>Others</td>
<td>Loaded from the MSR.</td>
</tr>
</tbody>
</table>

MSR

See Figure 67 on page 1261.

Execution resumes at effective address 0x0000_0000_0000_0A00, possibly offset as specified in Figure 68.

7.5.14 System Call Interrupt

A System Call interrupt occurs when a System Call instruction is executed.

The following registers are set:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction following the System Call instruction.</td>
</tr>
<tr>
<td>SRR1</td>
<td>33:36 Set to 0. 42:43 Set to indicate the LEV value specified by the System Call instruction that caused the interrupt, as follows.</td>
</tr>
<tr>
<td>Others</td>
<td>Loaded from the MSR.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEV</th>
<th>SRR1 42:43</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3*</td>
<td>undefined</td>
</tr>
<tr>
<td>*</td>
<td>reserved LEV value</td>
</tr>
</tbody>
</table>

Execution resumes at effective address 0x0000_0000_0000_0C00, possibly offset as specified in Figure 68.

Programming Note

Because the value of MSR$_{EE}$ is always 1 when the thread is in problem state, the simpler expression

$$(\text{MSR}_E \lor \neg (\text{MSR}_H)) \& \text{HDICE}$$

is equivalent to the expression given above.
### 7.5.15 Trace Interrupt

A Trace interrupt occurs when no higher priority exception exists and any instruction except `rfid`, `hrfid`, `urfid`, `rfscv`, or a Power-Saving Mode instruction is successfully completed, provided any of the following is true:

- the instruction is `mtmsr[d]` and MSRTE=0b10 when the instruction was initiated,
- the instruction is not `mtmsr[d]` and MSRTE=0b10,
- the instruction is a Branch instruction and MSRTE=0b01, or
- a CIABR match occurs.

Successful completion for an instruction means that the instruction caused no other interrupt. Thus a Trace interrupt never occurs for a System Call or System Call Vectored instruction, or for a Trap instruction that traps. The instruction that causes a Trace interrupt is called the “traced instruction”.

The following registers are set:

**SRR0**  
Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

**SRR1**  
- 33  
Set to 1.  
- 34  
Set to 0 if the traced instruction is a word instruction and to 1 if the traced instruction is a prefixed instruction.  
- 35  
Set to 1 if the Trace interrupt is not the result of a CIABR match and the traced instruction is a Load instruction other than a Load String instruction with string length of 0 or is specified to be treated as a Load instruction; otherwise set to 0.  
- 36  
Set to 1 if the Trace interrupt is not the result of a CIABR match and the traced instruction is a Store instruction other than a Store String instruction with string length of 0 or is specified to be treated as a Store instruction; otherwise set to 0.

**SIAR**  
For all Trace interrupts other than those caused by a CIABR match, set to the effective address of the traced instruction; for Trace interrupts caused by a CIABR match, not modified if MSRTE=0b00; otherwise undefined.

**SDAR**  
For all Trace interrupts other than those caused by a CIABR match, set to the effective address of the storage operand (if any) of the traced instruction; for Trace interrupts caused by a CIABR match, not modified if MSRTE=0b00; otherwise undefined.

If the state of the Performance Monitor is such that the Performance Monitor may be altering the SIAR and SDAR (i.e., if MMCR0PMAE=1), the contents of the SIAR and SDAR are undefined for the Trace interrupt and may change even when no Trace interrupt occurs.

**MSR**  
See Figure 67 on page 1261.

Execution resumes at effective address 0x0000_0000_0000_00D0, possibly offset as specified in Figure 68. For a Trace interrupt resulting from execution of an instruction that modifies the value of MSRIR, MSRD, MSRHV, LPCRAIL, or LPCRHAIL, the Trace interrupt vector location is based on the modified values.

---

**Programming Note**  
An attempt to execute an `sc` instruction with LEV=1 or LEV=2 in problem state, or an attempt to execute an `sc` instruction with LEV=2 in privileged non-hypervisor state, should be treated as a programming error.

An attempt to execute an `sc` instruction with LEV=2 when SMFCTRLE=0 should be treated as a programming error.

---

**Programming Note**  
An attempt to execute an `sc` instruction with LEV=1 or LEV=2 in problem state, or an attempt to execute an `sc` instruction with LEV=2 in privileged non-hypervisor state, should be treated as a programming error.

---

**Programming Note**  
Bit 33 is set to 1 for historical reasons.

---

**Programming Note**  
The following instructions are not traced.

- `rfid`
- `hrfid`
- `urfid`
- `rfscv`
- `sc`, `scv`, and Trap instructions that trap
- Power-Saving Mode instructions
- other instructions that cause interrupts (other than Trace interrupts)
- the first instructions of any interrupt handler (applies to Branch and Single Step tracing; CIABR matches may still occur)
- instructions that are emulated by software

In general, interrupt handlers can achieve the effect of tracing these instructions.
7.5.16 Hypervisor Data Storage Interrupt (HDSI)

A Hypervisor Data Storage interrupt occurs when no higher priority exception exists, either the thread is not in hypervisor state or an unsupported MMU configuration has been found or the access has been prevented by a problem in partition-scoped Radix Tree translation, and either

(a) HPT translation is being performed, VPM=0, LPCR\textsubscript{KBV}=1, and a Virtual Storage Page Class Key Protection exception exists or

(b) HPT translation is being performed and either a PRTE, STEG, or PTEG access causes a Secure Memory Protection exception or the value of the expression \((\neg\text{MSRDR}) \land (\text{VPM} \lor \text{PRTEV} \lor \text{MSRDR})\) is 1, and a data access cannot be performed, or

(c) Radix Tree translation is being performed and either a page fault or an unsupported MMU configuration in the PATE (unsupported value of RTS or RPDS) occurs attempting to access the LPID=0 process table or partition-scoped translation other than for the LPID=0 process table prevents an access from being performed

for any of the following reasons that can occur in the respective translation state.

- HR=0, data address translation is enabled (MSR\textsubscript{DR}=1) and the virtual address of any byte of the storage location specified by a \textit{Load}, \textit{Store}, \textit{icbi}, \textit{dcbz}, \textit{dcbst}, or \textit{dcbf} instruction cannot be translated to a real address because no valid PTE was found for the VPM translation.
- HR=1 and the guest real address of any byte of the storage location specified by a \textit{Load}, \textit{Store}, \textit{icbi}, \textit{dcbz}, \textit{dcbst}, or \textit{dcbf} instruction cannot be translated to a host real address because no valid PTE was found in the partition-scoped page table.
- The guest real address of a page directory entry, page table entry, or process table entry could not be translated when HR=1; or the virtual address of a process table entry or segment table entry group could not be translated when VPM=1 and HR=0.
- An unsupported MMU configuration is found. In addition to an unsupported radix tree configuration found for the partition-scoped tables, this type of exception will also be reported outside of hypervisor or ultravisor real mode for translation mode mismatches including UPRT=0 when HR=1, LPID=0 if MSR\textsubscript{HIV}=0 when HR=1, and HR=0 for LPID=0 when HR=1 for another partition ID. (Note that these conditions may not be detected until the associated values are about to cause a functional problem for the processor.)

- A reference or change bit update in a partition-scoped PTE cannot be performed (including for the process-scoped PDE or PTE or process table entry for a radix guest.

When reporting failure to set a reference or change bit for a table entry, whether the change bit must be set is inferred from whether the access is reported to be a store. (A load may report store if, when attempting to set the reference bit, the update of the change bit in the partition-scoped PTE mapping the process-scoped PTE fails.) Behavior is similar for access authority failures.

- HR=0, data address translation is disabled (MSR\textsubscript{DR}=0), and the virtual address of any byte of the storage location specified by a \textit{Load}, \textit{Store}, \textit{icbi}, \textit{dcbz}, \textit{dcbst}, or \textit{dcbf} instruction cannot be translated to a real address by means of the virtual real addressing mechanism.

The effective address specified by a \textit{lq}, \textit{stq}, \textit{lwat}, \textit{idat}, \textit{ilbarx}, \textit{ilharp}, \textit{ilwarx}, \textit{ldarx}, \textit{lqarx}, \textit{stdat}, \textit{stbcx}, \textit{stbch}, \textit{stwcx}, \textit{stwcx}, or \textit{stqcx}, instruction refers to storage that is Write Through Required or Caching Inhibited; or the effective address specified by a \textit{copy} or \textit{paste}. instruction refers to storage that is Caching Inhibited; or the effective address specified by a \textit{lwat}, \textit{idat}, \textit{stwat}, or \textit{stdat} instruction refers to storage that is Guarded.

- An accelerator is specified as the source of a \textit{copy} instruction or an attempt is made to access an accelerator that is not properly configured for the software’s use; HR=0 only.

The access violates storage protection. In addition to the legacy VPM cases (including those for Secure Memory Protection), this includes mismatches in access authority in which the process-scoped PTE permits the access but the partition-scoped PTE does not and Secure Memory Protection for a radix guest. It also includes lack of necessary authority for accesses to process-scoped tables (which cannot happen when HR=0), for example lack of write authority to set a reference bit in the process-scoped PTE (and Secure Memory Protection here as well). (In such a case, the “access” reported as failing would be the access to the process-scoped table. The
A Hypervisor Data Storage interrupt also occurs when no higher priority exception exists and an attempt is made to execute a Load Atomic or Store Atomic instruction specifying an invalid function code.

If a stbcx, sthcx, stwcx, stdcx, or stqcx instruction would not perform its store in the absence of a Hypervisor Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional Store to the specified effective address would cause a Hypervisor Data Storage interrupt, it is implementation-dependent whether a Hypervisor Data Storage interrupt occurs.

If the XER specifies a length of zero for an indexed Move Assist instruction, a Hypervisor Data Storage interrupt does not occur.

The following registers are set:

- **HSRR0**: Set to the effective address of the instruction that caused the interrupt.
- **HSRR1**
  - **33**: Set to 0.
  - **34**: Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.
  - **35:36**: Set to 0.
  - **42:47**: Set to 0.
  - **Others**: Loaded from the MSR.
- **HDSISR**
  - **32**: Set to 0.
  - **33**: Set to 1 if the translation for an attempted access is not found in the Page Table; otherwise set to 0.
  - **34:35**: Set to 0.
  - **36**: Set to 1 if the access is not permitted by Figure 45 or the read or read/write bits in Figure 46 as appropriate; otherwise set to 0.
  - **37**: Set to 1 if the access is due to a lq, stq, lwat, ldat, ldarx, lwarx, lwrcx, lwqcx, lqcx, stwat, stdat, stbcx, sthcx, stwcx, stdcx, or stqcx instruction that addresses storage that is Write Through Required or Caching Inhibited; or if the access is due to a copy or paste instruction that addresses storage that is caching inhibited; or if the access is due to a lwat, ldat, stwat, or stdat instruction that addresses storage that is Guarded; otherwise set to 0.
  - **38**: Set to 1 by an explicit access for a Store, dcbz, or Load/Store Atomic instruction; set to 1 when a process-scoped PTE update fails due to a lack of write authority or the inability to set the change bit in the partition-scoped PTE; otherwise set to 0.
  - **39:40**: Set to 0.
  - **41**: Set to 1 if a Data Address Watchpoint match occurs; otherwise set to 0.
  - **42**: Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.
  - **43**: Set to 1 if the access is not permitted by Secure Memory Protection; otherwise set to 0.
  - **44**: Set to 1 if an unsupported MMU configuration is found during the translation process.
  - **45**: Set to 1 if an attempt to atomically set a reference or change bit fails; otherwise set to 0.

Programming Note

When an attempt to execute a Load Atomic or Store Atomic instruction containing an invalid function code (see Figures 3 and 4 in Book II) causes an HDSI, the condition is very similar to an invalid form of an instruction. As a result, this instance of an HDSI makes to atomically set Reference and Change bit updates.

The number of attempts hardware makes to atomically set reference and change bits before triggering this exception is implementation dependent. The POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 6.7.12.
tion-caused exceptions. See Section 4.4, “Copy-Paste Facility”, in Book II for details. Additional information may be retained by the platform if the accelerator is not properly configured.

61

Set to 1 if an attempt is made to execute a Load Atomic or Store Atomic instruction specifying an invalid function code; otherwise set to 0.

62:63

Set to 0.

**HDAR**

Set to the effective address or portion of the VPN of a storage element, or undefined, as described in the following list. The list should be read from the top down; the HDAR is set as described by the first item that corresponds to an exception that is reported in the HDSISR. For example, if a Load Word instruction causes a storage protection violation and a Data Address Watchpoint match (and both are reported in the HDSISR), the HDAR is set to the effective address of a byte in the first aligned doubleword for which access was attempted in the page that caused the exception.

- undefined, for Load Atomic or Store Atomic instruction specifying an invalid function code
- undefined, when HDSISR60=1
- least significant 64 bits of the VA of the table entry or group when a process table entry or segment table entry group virtual address cannot be translated in Paravirtualized HPT mode with VPM=1.
- EA, when a Hypervisor Data Storage exception occurs for reasons other than a Data Address Watchpoint match
  - a byte in the block that caused the exception, for a Cache Management instruction
  - a byte in the first aligned doubleword for which access was attempted in the page that caused the exception, for a Load or Store instruction (“first” refers to address order; see Section 7.7).
- the first byte of overlap between the operand and the matching watched range, for a Data Address Watchpoint match

For the cases in which the HDAR is specified above to be set to an effective address, if the interrupt occurs in 32-bit mode the high-order 32 bits of the HDAR are set to 0.

**ASDR**

When HR=0, loaded with VSID, B, Ks, Kp, N, C, L, and LP values from the segment descriptor that translated the access or indicated the base of the table, or undefined, as described in the following list. When HR=1 (nested translation is taking place), loaded with the guest real address down to bit 51 of a storage element or table entry, or undefined, as described in the following list. The list should be read from the top down; the ASDR is set as described by the first item that corresponds to an exception that is reported in the HDSISR.

- undefined, for Load Atomic or Store Atomic instruction specifying an invalid function code
- undefined, when HDSISR60=1
- the guest real page address of the table entry when a process table or process-scoped page directory or page table entry guest real address cannot be translated or the VSID of the table entry (group) when a Process Table Entry or Segment Table Entry Group virtual address cannot be translated (the rest of the segment descriptor is implied, or, for the base page size, comes from the Partition Table Entry or Process Table Entry, respectively).
- the guest real address of the process-scoped PDE or PTE or process table entry when a reference or change bit in the partition-scoped PTE mapping the process-scoped PDE or PTE or process table entry cannot be set atomically
- the guest real address of the storage element when a reference or change bit in the partition-scoped PTE cannot be set atomically
- the guest real address of the storage element, process table entry, page directory entry, or page table entry (depending on which partition-scoped table has the flaw) for an unsupported radix tree configuration for the partition-scoped tables (the effective address for other cases of the invalid MMU configuration exception is found in the HDAR)
- the guest real address of the process-scoped PTE when an attempt is
made to set a reference or change bit without write authority in the partition-scoped PTE that maps it

- the guest real address or segment descriptor associated with the specified storage element when a Hypervisor Data Storage exception occurs for reasons other than a Data Address Watchpoint match
- undefined, for a Data Address Watchpoint match, unsupported MMU configuration, or accesses to storage that is Caching Inhibited or Write Through Required by the instructions that are prohibited from making such accesses.

If multiple Hypervisor Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the HDSISR. If the HDSISR reports other exceptions together with a Virtualized Page Class Key Storage Protection exception that occurs when LPCRKBV=1 and Virtualized Partition Memory is disabled by VPM=0, the other exceptions are actually DSIs.

## Programming Note

A Virtual Page Class Key Storage Protection exception that occurs with LPCRKBV=1 and Virtualized Partition Memory disabled by VPM=0 identifies an access that must be emulated by the hypervisor. When it is reported together with other exceptions in the HDSISR, the hypervisor should service the Virtual Page Class Key Storage Protection exception first. This is in part because the operating system may be using some PTE fields for non-architected purposes, which could in turn cause spurious exceptions to be reported.

Execution resumes at effective address 0x0000_0000_0000_0E00, possibly offset as specified in Figure 68.

### 7.5.17 Hypervisor Instruction Storage Interrupt (HISI)

A Hypervisor Instruction Storage interrupt occurs when no higher priority exception exists, either the thread is not in hypervisor state or an unsupported MMU configuration has been found or the access has been prevented by a problem in partition-scoped Radix Tree translation, and either

(a) HPT translation is being performed and either a PRTE, STEG, or PTEG access causes a Secure Memory Protection exception or the value of the expression (~MSRIR) | (VPM & PRTEV & MSRIR) is 1, and the next instruction to be executed cannot be fetched, or

(b) Radix Tree translation is being performed and either a page fault or an unsupported MMU configuration in the PATE (unsupported value of RTS or RPDS) occurs attempting to access the LPID=0 process table or partition-scoped translation other than for the LPID=0 process table prevents the next instruction to be executed from being fetched

for any of the following reasons that can occur in the respective translation state.

- Instruction address translation is enabled (MSRIR=1) and the virtual address cannot be translated to a real address because no valid PTE was found for the VPM translation.
- HR=1 and the guest real address of the instruction cannot be translated to a host real address because no valid PTE was found in the partition-scoped page table.
- The guest real address of a page directory entry or process table entry could not be translated when HR=1; or the virtual address of a process table entry or segment table entry group could not be translated when VPM=1 and HR=0.
- An unsupported MMU configuration is found. In addition to an unsupported radix tree configuration found for the partition-scoped tables, this type of exception will also be reported outside of hypervisor or ultravisor real mode for translation mode mismatches including UPRT=0 when HR=1, LPID=0 if MSRIRV=0 when HR=1, and HR=0 for LPID=0 when HR=1 for another partition ID. (Note that these conditions may not be detected until the associated values are about to cause a functional problem for the processor.)
- A reference or change bit update in a partition-scoped PTE cannot be performed (including for the process-scoped PDE or PTE or process table entry for a radix guest.
- HR=0, instruction address translation is disabled (MSRIR=0), and the virtual address cannot be translated to a real address by means of the virtual real addressing mechanism.
- The fetch violates storage protection. In addition to the legacy VPM cases (including those for Secure Memory Protection), this includes mismatches in access authority in which the process-scoped PTE permits the access but the partition-scoped PTE does not and Secure Memory Protection for a radix guest. It also includes lack of necessary authority for accesses to process-scoped tables (which cannot happen when HR=0), for example lack of write authority to set a reference bit in the process-scoped PTE (and Secure Memory Protection here as well). (In such a case, the “access” reported as failing would be the access to the process-scoped table. The
ASDR would provide the guest real address of the table entry.)

- The instruction is a prefixed instruction and is in storage that is Caching Inhibited.

The following registers are set:

**HSRR0**
- Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, HSRR0 is set to the branch target address).

**HSRR1**
- **33** Set to 1 if the translation for an attempted access is not found in the Page Table; otherwise set to 0.
- **34** Set to 0.
- **35** Set to 1 if the access is to No-execute (as indicated by the N bit in the segment table entry and HPT PTE or the exec bit in the EAA field of the Radix PTE) or Guarded storage, or is to Caching Inhibited storage and is for a prefixed instruction; otherwise set to 0.
- **36** Set to 1 if the access is not permitted by Figure 45 or the execute bit in Figure 46 as appropriate; otherwise set to 0.
- **42** Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.
- **43** Set to 1 if the access is not permitted by Secure Memory Protection; otherwise set to 0.
- **44** Set to 1 if an unsupported MMU configuration is found during the translation process.
- **45** Set to 1 if an attempt to atomically set a reference or change bit fails; otherwise set to 0.

**Others**
- Loaded from the MSR.

**HDAR**
- Set to the least significant 64 bits of the VA of a table entry or group when HR=0 and a process table entry or segment table entry group virtual address cannot be translated and VPM=1. May be set spuriously in other cases.

**ASDR**
- When HR=0, loaded with VSID, B, Ks, Kp, N, C, L, and LP values from the segment descriptor that translated the access or indicated the base of the table, or undefined, as described in the following list. When HR=1 (nested translation is taking place), loaded with the guest real address down to bit 51 of the instruction or table entry, or undefined, as described in the following list.

- the guest real address of the table entry when a process table or process-scoped page directory or page table entry guest real address cannot be translated or the VSID of the table entry (group) when a Process Table Entry or Segment Table Entry Group virtual address cannot be translated (the rest of the segment descriptor is implied, or, for the base page size, comes from the Partition Table Entry or Process Table Entry, respectively).
- the guest real address of the process-scoped PDE or PTE or process table entry when a reference or change bit in the partition-scoped PTE mapping the process-scoped PDE or PTE or process table entry cannot be set atomically
- the guest real address of the instruction when a reference or change bit in the partition-scoped PTE cannot be set atomically
- the guest real address of the instruction, process table entry, page directory entry, or page table entry (depending on which partition-scoped table has the flaw) for an unsupported radix tree configuration for the partition-scoped tables (the effective address for other cases of the invalid MMU configuration exception will be found in HSRR0)
- the guest real address of the process-scoped PTE when an attempt is made to set a reference bit without

**Programming Note**

The number of attempts hardware makes to atomically set reference and change bits before triggering this exception is implementation dependent. The POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 6.7.12.
write authority in the partition-scoped PTE that maps it
- the guest real address or segment descriptor associated with the instruction that the thread would have attempted to execute next if no interrupt conditions were present (partition-scoped page fault or protection exception)
- undefined for unsupported MMU configuration

**MSR** See Figure 67.

If multiple Hypervisor Instruction Storage exceptions occur due to attempting to fetch a single instruction, any one or more of the bits corresponding to these exceptions may be set to 1 in HSRR1.

Execution resumes at effective address 0x0000_0000_0000_0E10, possibly offset as specified in Figure 68.

### 7.5.18 Hypervisor Emulation Assistance Interrupt

A Hypervisor Emulation Assistance interrupt is generated when execution is attempted of an illegal instruction, or of a reserved instruction or an instruction that is not provided by the implementation. It is also generated under the following conditions.

- When MSRRHV PR=0b00 and LPCR EVIRT=1, execution is attempted of a hypervisor privileged instruction, or of a mtspr or mfspr instruction that specifies an SPR that is hypervisor privileged for the operation or that specifies PTCR, DAWRn, DAWRXn, or CIABR when those SPRs are ultravisor privileged for the operation.
- When MSRRHV PR = 0b010, execution is attempted of a mtspr or mfspr instruction that specifies PTCR, DAWRn, DAWRXn, or CIABR when those SPRs are ultravisor privileged for the operation.
- When MSR PR=1, execution is attempted of an mtspr or mfspr instruction that specifies SPR with spr0=0 that is not provided by the implementation.
- When MSR PR=0, execution is attempted of an mtspr or mfspr instruction that specifies SPR 0, 4, 5, or 6.
- When MSR PR=0 and LPCREVIRT=1, execution is attempted of an mtspr or mfspr instruction that specifies an SPR other than 0, 4, 5, or 6 that is not provided by the implementation.

A Hypervisor Emulation Assistance interrupt may be generated when execution is attempted of an instruction that is in invalid form or that is treated as if the instruction form were invalid.

The following registers are set:

**HSRR0** Set to the effective address of the instruction that caused the interrupt.

**HSRR1**
- 33 Set to 0.
- 34 Set to 0 if the instruction that caused the interrupt is a word instruction (or a prefixed instruction when prefixed instructions are unavailable based on the PCR setting) and to 1 if the instruction that caused the interrupt is a prefixed instruction.
- 35:36 Set to 0.
- 42:44 Set to 0.
- 45 Set to 1 for an attempt, when MSRRHV PR = 0b00 and LPCR EVIRT=1, to execute a hypervisor privileged instruction or an mtspr or mfspr instruction that specifies an SPR that is hypervisor privileged for the operation or that specifies PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor privileged for the operation, or for an attempt when MSRRHV PR = 0b010 to execute an mtspr or mfspr instruction that specifies PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor privileged for the operation; otherwise set to 0.
- 46:47 Set to 0.
- Others Loaded from the MSR.

**MSR** See Figure 67 on page 1261.

**HEIR** Set to a copy of the instruction that caused the interrupt.

If the interrupt is caused by an attempt to execute an invalid form of a hypervisor privileged instruction when MSRRHV PR = 0b00 and LPCR EVIRT=1, it is implementation dependent whether HSRR145 is set to 0 (reflecting the invalid instruction form) or to 1 (reflecting the privilege violation).

Execution resumes at effective address 0x0000_0000_0000_0E40, possibly offset as specified in Figure 68.
This Programming Note illustrates how Hypervisor Emulation Assistance interrupts should be handled by software, including in environments that support nested hypervisors. For simplicity, this Programming Note ignores effects of the SMF facility (equivalently, assumes that SMFCtrlLE=0).

In this Note, “the hypervisor” may be the hypervisor to which hardware passes control when a Hypervisor Emulation Assistance interrupt occurs or, in an environment that supports nested hypervisors, may be a nested hypervisor. The hypervisor to which hardware passes control when a Hypervisor Emulation Assistance interrupt occurs is here called the “level 0 hypervisor,” and is the only level of hypervisor that runs with MSR HV PR=0b10 and that can access hypervisor resources directly; nested hypervisors run with MSR HV PR=0b00 and their attempts to access hypervisor resources are virtualized by a higher-level hypervisor as described below. In this Note, the hypervisor receiving the Hypervisor Emulation Assistance interrupt (which may have been passed from a higher-level hypervisor as described below) is called the “level N hypervisor.” This Note assumes that LPCREVIRT=1 if nested hypervisors are used. (A Hypervisor Emulation Assistance interrupt can set HSRR145 to 1 only when LPCREVIRT=1.) Higher level numbers correspond to lower level hypervisors.

In the description immediately below, it is assumed that nested hypervisors (if any) are new versions of the existing hypervisor, and that the purpose of the nesting is to test the nested hypervisors before using them as level 0 hypervisors.

When a Hypervisor Emulation Assistance interrupt is received by the level N hypervisor, the cases and their suggested handling are as follows.

- The program that caused the interrupt is the level N hypervisor itself.
  - HSRR145=0: Emulate the instruction, recover from the error, or terminate this hypervisor, as appropriate.
  - HSRR145=1: Cannot occur for N=0; will not occur for N>0 if the hypervisor nesting software is written correctly.

- The program that caused the interrupt is not the level N hypervisor.
  - The program most recently dispatched by the level N hypervisor is a program.
    - HSRR145=0: Pass control to the level N+1 hypervisor as if the instruction had caused a Hypervisor Emulation Assistance interrupt (with HSRR145=0) to that hypervisor.
    - HSRR145=1: Either terminate the operating system or pass control to the operating system as if the instruction had caused an “Illegal Instruction type Program interrupt” as described in a Programming Note near the end of Section 7.5.9.
  - The program most recently dispatched by the level N hypervisor is an application program.
    - HSRR145=0: Emulate the instruction if appropriate; otherwise terminate the application program.
    - HSRR145=1: Cannot occur.

The preceding description implicitly assumes that any nested hypervisors being tested will, when run at level 0, be run on processors that support the same version of the architecture as the processor on which they are being tested. If instead they will be run on processors that support a newer version of the architecture, the level 0 hypervisor should behave as described above if the interrupt is caused by an instruction that is unchanged between the two architecture versions. However, if the interrupt is caused by an instruction that differs between the two architecture versions (e.g., an instruction that is added by the newer version of the architecture), the level 0 hypervisor should emulate the behavior of the newer processor, rather than, for example, passing the interrupt to a level 1 hypervisor.

Other uses of nested hypervisors are also possible. For example, software that is designed to interact, nearly simultaneously, with the hypervisor instance that is running on each of many processors could be tested on a single processor by running multiple level 1 hypervisors under a single level 0 hypervisor.

It is expected that in practice there will be at most two levels of nested hypervisor (i.e., N≤2). (For example, two levels are needed in the case described in detail above, to test the ability of the nested hypervisors at level 1 to support nested hypervisors.)
7.5.19 Hypervisor Maintenance Interrupt

A Hypervisor Maintenance interrupt occurs when no higher priority exception exists, a Hypervisor Maintenance exception exists (a bit in the HMER is set to one), the exception is enabled in the HMEER, and the value of the following expression is 1.

\[(MSREE \mid \neg(MSRHV))\]

The following registers are set:
- **HSRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **HSRR1**: Settings depend on the situation.
- **Others**: Loaded from the MSR.

If a Hypervisor Emulation Assistance interrupt occurs with HSRR1_{45}=0 when the thread is not in hypervisor state, for an instruction that the hypervisor does not emulate, the hypervisor should pass control to the operating system as if the instruction had caused an "Illegal Instruction type Program interrupt", as described in a Programming Note near the end of Section 7.5.9, “Program Interrupt” on page 1272.

Similarly, if a Hypervisor Emulation Assistance interrupt occurs with HSRR1_{45}=1 when the thread is in privileged non-hypervisor state, for an instruction that the hypervisor does not virtualize, the hypervisor should pass control to the operating system as if the instruction had caused a Privileged Instruction type Program interrupt, as described in another Programming Note near the end of Section 7.5.9, “Program Interrupt” on page 1272.

The exception bits in the HMER are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an **mthmer** instruction.

Execution resumes at effective address 0x0000_0000_0000_0E60.

7.5.20 Directed Hypervisor Doorbell Interrupt

A Directed Hypervisor Doorbell interrupt occurs when no higher priority exception exists, a Directed Hypervisor Doorbell exception is present, and the value of the following expression is 1.

\[(MSREE \mid \neg(MSRHV))\]

Directed Hypervisor Doorbell exceptions are generated when Directed Hypervisor Doorbell messages (see Chapter 11) are received and accepted by the thread.

The following registers are set:
- **HSRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **HSRR1**: Settings depend on the situation.
- **Others**: Loaded from the MSR.

**Programming Note**

If an implementation uses the HMER to record that a readable resource, such as the Time Base, has been corrupted, then, because the HMI is disabled in the hypervisor state, it is necessary for the hypervisor to check HMER after reading that resource to be sure an error has not occurred.

**Programming Note**

Because the value of MSREE is always 1 when the thread is in problem state, the simpler expression

\[(MSRE\mid \neg(MSRHV))\]

is equivalent to the expression given above.
7.5.21 Hypervisor Virtualization Interrupt

A Hypervisor Virtualization interrupt occurs when no higher priority exception exists, a Hypervisor Virtualization exception exists, and the value of the following equation is 1.

\[(\text{MSREE} \lor \neg (\text{MSRHV}) \lor \text{MSRPR}) \land \text{HVICE}\]

The occurrence of the interrupt does not cause the exception to cease to exist.

**HSRR0** Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

**HSRR1**
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

**MSR** See Figure 67 on page 1261.

Execution resumes at effective address 0x0000_0000_0000_0E80, possibly offset as specified in Figure 68.

Programming Note
Because the value of MSREE is always 1 when the thread is in problem state, the simpler expression \((\text{MSREE} \lor \neg (\text{MSRHV}))\) is equivalent to the expression given above.

7.5.23 Vector Unavailable Interrupt

A Vector Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a Vector instruction (including Vector loads, stores, and moves), and MSRVEC=0.

The following registers are set:

**SRR0** Set to the effective address of the instruction that caused the interrupt.

**SRR1**
- **33** Set to 0.
- **34** Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.
- **35:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

**MSR** See Figure 67 on page 1261.

Execution resumes at effective address 0x0000_0000_0000_0F20, possibly offset as specified in Figure 68.

7.5.24 VSX Unavailable Interrupt

A VSX Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a VSX instruction (including VSX loads, stores, and moves), and MSRVSX=0.

The following registers are set:

**SRR0** Set to the effective address of the instruction that caused the interrupt.

**SRR1**
- **33** Set to 0.
- **34** Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.
- **35:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

**MSR** See Figure 67 on page 1261.

Execution resumes at effective address 0x0000_0000_0000_0F00, possibly offset as specified in Figure 68.
7.5.25 Facility Unavailable Interrupt

A Facility Unavailable interrupt occurs when no higher priority exception exists, and one of the following occurs.
- A facility is accessed in problem state when it has been made unavailable by the FSCR.
- A Performance Monitor register is accessed or a \texttt{clbrb} or \texttt{mfbrbe} instruction is executed in problem state when it has been made unavailable by MMCR0.

The following registers are set:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{SRR0}</td>
<td>Set to the effective address of the instruction that caused the interrupt.</td>
</tr>
<tr>
<td>\texttt{SRR1}</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>33</td>
<td>Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.</td>
</tr>
<tr>
<td>34</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>35:36</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>42:47</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>Others</td>
<td>Loaded from the MSR.</td>
</tr>
</tbody>
</table>

Execution resumes at effective address 0x0000_0000_0000_0F40, possibly offset as specified in Figure 68.

7.5.26 Hypervisor Facility Unavailable Interrupt

A Hypervisor Facility Unavailable interrupt occurs when no higher priority exception exists, and one of the following occurs.
- A facility is accessed in problem or privileged non-hypervisor states when it has been made unavailable by the HFSCR.
- The \texttt{stop} instruction is executed in privileged non-hypervisor state when any of the following conditions exist.
  - $\text{PSSCR}_{\text{MTL}} > \text{PSSCR}_{\text{PSLL}}$
  - $\text{PSSCR}_{\text{RL}} > \text{PSSCR}_{\text{PSLL}}$

The following registers are set:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{HSRR0}</td>
<td>Set to the effective address of the instruction that caused the interrupt.</td>
</tr>
<tr>
<td>\texttt{HSRR1}</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>33</td>
<td>Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.</td>
</tr>
<tr>
<td>34</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>35:36</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>42:47</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>Others</td>
<td>Loaded from the MSR.</td>
</tr>
</tbody>
</table>

Execution resumes at effective address 0x0000_0000_0000_0F60, possibly offset as specified in Figure 68.
The Hypervisor Facility Unavailable interrupt handler should either (a) make the facility, the attempted use of which caused the interrupt, available, or (b) pass control to the operating system as if the instruction that caused the interrupt had instead caused an "Illegal Instruction type Program interrupt", as described in a Programming Note near the end of Section 7.5.9. Specifically, for choice (b) the hypervisor should pass control to the operating system at the operating system’s Program interrupt vector location, with all registers (SRR0, SRR1, MSR, GPRs, etc.) set as if the instruction had caused a Privileged Instruction type Program interrupt, except with SRR1_{44:45} set to 0b10. (This behavior is the same as that provided by the Hypervisor Emulation Assistance interrupt handler when that interrupt is caused by an illegal instruction or by \textit{mt/fspr} specifying an undefined SPR number.) In general this behavior by the Hypervisor Facility Unavailable interrupt handler provides to the operating system the appearance that the instructions in the facility are illegal instructions and that the SPRs in the facility correspond to undefined SPR numbers. The cases in which it does not provide this appearance are as follows.

1. **privileged instruction executed in problem state**
   Because Privileged Instruction type Program interrupt has higher priority than Hypervisor Facility Unavailable interrupt, an attempt in problem state to execute a privileged instruction made unavailable by the HFSCR will cause a Privileged Instruction type Program interrupt to the operating system, rather than a Hypervisor Facility Unavailable interrupt, so the hypervisor will not have opportunity to make the instruction appear to be illegal. (It may be useful to note that the handling described in choice (b) above together with the behavior of this case provides behavior, in problem state, that is equivalent to the behavior that would be obtained by making the facility unavailable by means of the PCR.)

2. **\textit{mt/fspr} executed in privileged non-hypervisor state when LPCREVIRT=0**
   \textit{mt/fspr} specifying an undefined SPR number (other than 0, 4, 5, 6) and executed in privileged non-hypervisor state when LPCREVIRT=0 will be treated as a no-op. If instead the SPR number is defined and the SPR is made unavailable by the HFSCR a Hypervisor Facility Unavailable interrupt will occur, and there is no easy way for the interrupt handler to determine that the interrupting instruction is \textit{mt/fspr} and hence should be treated as a no-op. (Hypervisor Facility Unavailable interrupt does not set HEIR.) Passing control to the operating system’s Program interrupt handler in the manner described above is preferable to incurring the software complexity and performance cost of emulating the no-op behavior.

### 7.5.27 System Call Vectored Interrupt

A System Call Vectored interrupt occurs when a System Call Vectored instruction is executed.

The following registers are set:

- **LR**
  Set to the effective address of the instruction following the System Call Vectored instruction.

- **CTR**
  - 33:36: undefined
  - 42:47: undefined

- **Others**
  Loaded from corresponding bits of the MSR.

- **MSR**
  See Figure 67 on page 1261.

Execution resumes at the effective address specified in Figure 68.
7.5.28 Directed Ultravisor Doorbell Interrupt

A Directed Ultravisor Doorbell interrupt occurs when no higher priority exception exists, SMFCTRL\textsubscript{LE}=1, a Directed Ultravisor Doorbell exception is present, and the value of the following expression is 1.

\[(\text{MSR}_{\text{EE}} \mid \neg (\text{MSR}_{\text{S HV PR}}=0b110))\]

Directed Ultravisor Doorbell exceptions are generated when Directed Ultravisor Doorbell messages (see Chapter 11) are received and accepted by the thread.

The following registers are set:

- **USRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **USRR1**: Set to 0.
  - 33:36: Set to 0.
  - 42:47: Set to 0.
  - Others: Loaded from the MSR.

**MSR**: See Figure 67 on page 1261.

Execution resumes at effective address 0x0000_0000_0000_0FA0.

7.6 Partially Executed Instructions

If a Data Storage, Data Segment, Alignment, system-caused, or imprecise exception occurs while a Load or Store instruction is executing, the instruction may be aborted. In such cases the instruction is not completed, but may have been partially executed in the following respects.

- Some of the bytes of the storage operand may have been accessed, except that if access to a given byte of the storage operand would violate storage protection, that byte is neither copied to a register by a Load instruction nor modified by a Store instruction. Also, the rules for storage accesses given in Section 6.8.1, “Guarded Storage” and in Section of Book II are obeyed.

- Some registers may have been altered as described in the Book II section cited above.

- Reference and Change bits may have been updated as described in Section 6.7.12.

- For a \textit{stbcx.}, \textit{sthcx.}, \textit{stwcx.}, \textit{stdcx.}, or \textit{stqcx.} instruction that is executed in-order, CR0 may have been set to an undefined value and the reservation may have been cleared.

The architecture does not support continuation of an aborted instruction but intends that the aborted instruction be re-executed if appropriate.

---

### Programming Note

An exception may result in the partial execution of a \textit{Load} or \textit{Store} instruction. For example, if the Page Table Entry that translates the address of the storage operand is altered, by a program running on another thread, such that the new contents of the Page Table Entry preclude performing the access, the alteration could cause the \textit{Load} or \textit{Store} instruction to be aborted after having been partially executed.

As stated in the Book II section cited above, if an instruction is partially executed the contents of registers are preserved to the extent that the instruction can be re-executed correctly. The consequent preservation is described in the following list. For any given instruction, zero, one, or two items in the list apply.

- For a fixed-point \textit{Load} instruction that is not a multiple or string form, if RT=RA or RT=RB then the contents of register RT are not altered.
- For an \textit{lq} instruction, if RT+1 = RA then the contents of register RT+1 are not altered.
- For an update form \textit{Load} or \textit{Store} instruction, the contents of register RA are not altered.
7.7 Exception Ordering

Since multiple exceptions can exist at the same time and the architecture does not provide for reporting more than one interrupt at a time, the generation of more than one interrupt is prohibited. Some exceptions, such as the Mediated External exception, persist and can be deferred. However, other exceptions would be lost if they were not recognized and handled when they occur. For example, an External interrupt was generated when a Data Storage exception existed, the Data Storage exception would be lost. If the Data Storage exception was caused by a Store Multiple instruction for which the storage operand crosses a virtual page boundary and the exception was a result of attempting to access the second virtual page, the store could have modified locations in the first virtual page even though it appeared that the Store Multiple instruction was never executed.

For the above reasons, all exceptions are prioritized with respect to other exceptions that may exist at the same instant to prevent the loss of any exception that is not persistent. Some exceptions cannot exist at the same instant as some others.

Data Storage, Hypervisor Data Storage, Data Segment, and Alignment exceptions occur as if the storage operand were accessed one byte at a time in order of increasing effective address (with the obvious caveat if the operand includes both the maximum effective address and effective address 0). (The required ordering of exceptions on components of non-atomic accesses does not extend to the performing of the component accesses in the event of an exception. For example, if byte n causes a data storage exception, it is not necessarily true that the access to byte n-1 has been performed.)

7.7.1 Unordered Exceptions

With one exception, the exceptions listed here are unordered, meaning that they may occur at any time regardless of the state of the interrupt processing mechanism. These exceptions are recognized and processed when presented. The exception is that a Machine Check caused by an attempt to access an accelerator as other than an operand of copy or paste is ordered similarly to the corresponding type of storage access exception. (Note that this results in two different orderings. The one for instruction fetch occurs early, as item 2 in the "Instruction-Caused and Precise" list. The one for data access appears later, within case 3 of the appropriate "Function-Dependent" listings.)

1. System Reset
2. Machine Check except for those caused by an invalid attempt to access an accelerator

7.7.2 Ordered Exceptions

The exceptions listed here are ordered with respect to the state of the interrupt processing mechanism. With one exception, in the following list the hypervisor forms of the Data Storage and Instruction Storage exceptions can be substituted for the non-hypervisor forms since the hypervisor forms cannot be caused by the same instruction and have the same ordering. The exception is that Virtual Page Class Key Storage Protection exceptions that occur when LPCRKBV=1 and Virtualized Partition Memory is disabled by VPM=0 cause only a Hypervisor Data Storage exception (and never a Data Storage exception).

In the list below for Instruction-Caused and Precise exceptions, for prefixed instructions items 1, 2, and 6 apply to the first word of the instruction, and determination of whether the exceptions of items 3-5 and 7 occur is based on the first word of the instruction. (Items 3-7 can occur only for prefixed instructions.)

Programming Note

For a prefixed instruction there is no need to determine whether the exceptions of items 1, 2, and 6 can be caused by the second word of the instruction.

- If the instruction is correctly aligned (i.e., is at an effective address that is not equal to 60 modulo 64), the determination would be the same for the second word as for the first word.
- If the instruction is not correctly aligned, an exception will occur due to one of items 1-7 applied to the first word, and the corresponding interrupt obviates the need for hardware to access the second word.

System-Caused or Imprecise

1. Program
   - Imprecise Mode Floating-Point Enabled Exception
2. Directed Ultravisor Doorbell
3. Hypervisor Maintenance

Instruction-Caused and Precise

1. Instruction Segment
2. [Hypervisor] Instruction Storage other than case described in item 6
3. Hypervisor Emulation Assistance due to PCR making all prefixed instructions unavailable
4. Hypervisor Facility Unavailable due to HFSCR making all prefixed instructions unavailable
5. Facility Unavailable due to HFSCR making all prefixed instructions unavailable
6. [Hypervisor] Instruction Storage for prefixed instruction in Caching Inhibited storage
7. Alignment for incorrectly aligned prefixed instruction
For implementations that execute multiple instructions in parallel using pipeline or superscalar techniques, or combinations of these, it can be difficult to understand the ordering of exceptions. To understand this ordering it is useful to consider a model in which each instruction is fetched, then decoded, then executed, all before the next instruction is fetched. In this model, the exceptions a single instruction would generate are in the order shown in the list of instruction-caused exceptions. Exceptions with different numbers have different ordering. Exceptions with the same numbering but different lettering are mutually exclusive and cannot be caused by the same instruction. The Hypervisor Virtualization, External, [Hypervisor] Decrementer, Performance Monitor, Directed Privileged Doorbell, and Directed Hypervisor Doorbell interrupts have equal ordering. Similarly, where Data Storage, Data Segment, and Alignment exceptions are listed in the same item, and where Hypervisor Emulation Assistance and Privileged Instruction exceptions are listed in the same item, they have equal ordering.

Even on threads that are capable of executing several instructions simultaneously, or out of order, instruction-caused interrupts (precise and imprecise) occur in program order.

---

**Programming Note**

Despite that debug address matches are EA based, the exceptions they cause are not necessarily ordered before translation-caused exceptions. For example, it may be considered advantageous to take a page fault that would have prevented an access rather than a DAWR match exception.

---

### 7.8 Event-Based Branch Exception Ordering

Event-based exceptions are not ordered because they can occur simultaneously. Whenever an event-based exception occurs and the exception is enabled, the corresponding “exception occurred” bit in the BESCR is set to 1. See Section 6.2.1 of Book II.

### 7.9 Interrupt Priorities

This section describes the relationship of nonmaskable, maskable, precise, and imprecise interrupts. In the following descriptions, the interrupt mechanism waiting for all possible exceptions to be reported includes only exceptions caused by previously initiated instructions (e.g., it does not include waiting for the Decrementer to step through zero). The exceptions are listed in order of highest to lowest priority. The phrase “corresponding interrupt” means the interrupt
having the same name as the exception unless the thread is in power-saving mode, in which case the phrase means the System Reset interrupt.

Unless otherwise stated or obvious from context, it is assumed below that one of the following conditions is satisfied.

- The thread is not in power-saving mode and the interrupt, unless it is the Machine Check interrupt, is not disabled. (For the Machine Check interrupt no assumption is made regarding enablement.)
- The thread is in power-saving mode and the exception is enabled to cause exit from the mode.

With one exception, in the following list the hypervisor forms of the Data Storage and Instruction Storage exceptions can be substituted for the non-hypervisor forms since the hypervisor forms cannot be caused by the same instruction and have the same priority. The exception is that exceptions caused by Virtual Page Class Key Storage Protection exceptions that occur when LPCRKEYV=1 and Virtualized Partition Memory is disabled by VPM=0 cause only a Hypervisor Data Storage exception (and never a Data Storage exception).

1. System Reset

System Reset exception has the highest priority of all exceptions. If this exception exists, the interrupt mechanism ignores all other exceptions and generates a System Reset interrupt.

Once the System Reset interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

2. Machine Check

With one exception, the Machine Check exception is the second highest priority exception. If this exception exists and a System Reset exception does not exist, the interrupt mechanism ignores all other exceptions and generates a Machine Check interrupt. The exception is that a Machine Check caused by an attempt to access an accelerator as other than an operand of copy or paste, is prioritized similarly to the corresponding type of storage access exception. (Note that this results in two different priorities. The one for data access is higher, appearing as item e or f in the listings for different types of Load and Store instructions. The one for instruction fetch is lower, in category K.)

Once the Machine Check interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

3. Instruction-Caused and Precise

This exception is the third highest priority exception. When this exception is created, the interrupt mechanism waits for all possible Imprecise exceptions to be reported. It then generates the appropriate ordered interrupt if no higher priority exception exists when the interrupt is to be generated. Within this category a particular instruction may present more than a single exception. When this occurs, those exceptions are ordered in priority as indicated in the following lists. Where [Hypervisor] Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal priority (i.e., the hardware may generate any one of the three interrupts for which an exception exists).

A. Fixed-Point Loads and Stores
   a. These exceptions are mutually exclusive and have the same priority:
      - Hypervisor Emulation Assistance
      - Program - Privileged Instruction
   b. Hypervisor Facility Unavailable
   c. Facility Unavailable
   d. Data Storage for the case of Fixed-Point Load or Store Caching Inhibited instructions with MSRDR=1 or the case of an invalid function code for an Atomic Memory Operation
   e. all other Data Storage, Hypervisor Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment (other than for incorrectly aligned prefixed instruction)
   f. Trace

B. Floating-Point Loads and Stores
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. Floating-Point Unavailable
   d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment (other than for incorrectly aligned prefixed instruction)
   e. Trace

C. Vector Loads and Stores
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. Vector Unavailable
   d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment (other than for incorrectly aligned prefixed instruction)
   e. Trace

D. VSX Loads and Stores
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. VSX Unavailable
d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment (other than for incorrectly aligned prefixed instruction)
e. Trace

E. Other Floating-Point Instructions
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. Floating-Point Unavailable
   d. Program - Precise Mode Floating-Point Enabled Exception
   e. Trace

F. Other Vector Instructions
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. Vector Unavailable
   d. Trace

G. Other VSX Instructions
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. VSX Unavailable
   d. Program - Precise Mode Floating-Point Enabled Exception
   e. Trace

H. rfebb, rfscv, rfid, hrfid, urfid, and mtmsr[d]
   a. These exceptions are mutually exclusive and have the same priority:
      - Program - Privileged Instruction, for all except rfebb
      - Hypervisor Emulation Assistance, for rfebb, rfscv, hrfid and mtmsr
   b. Hypervisor Facility Unavailable (rfebb only)
   c. Facility Unavailable (rfebb only)
   d. Program - Floating-Point Enabled Exception, for all except rfebb
   e. Trace, for rfebb and mtmsr[d] only

I. Other Instructions
   a. These exceptions or groups of exceptions are mutually exclusive and have the same priority (the members of a group are not mutually exclusive, but have the same priority):
      - Program - Trap
      - System Call
      - System Call Vectored
      - Hypervisor Emulation Assistance or Program (Privileged Instruction)
   b. Hypervisor Facility Unavailable
   c. Facility Unavailable
   d. Trace

J. [Hypervisor] Instruction Storage, Instruction Segment, Alignment for incorrectly aligned prefixed instruction, and Machine Check for invalid accelerator access

These exceptions have the lowest priority in this category. They are recognized only when all instructions prior to the instruction causing one of these exceptions appear to have completed and that instruction is the next instruction to be executed. The four exceptions are mutually exclusive.

The priority of these exceptions is specified for completeness and to ensure that they are not given more favorable treatment. It is acceptable for an implementation to treat these exceptions as though they had a lower priority.

4. Program - Imprecise Mode Floating-Point Enabled Exception

This exception is the fourth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

5. Directed Ultravisor Doorbell

This exception is the fifth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

6. Hypervisor Maintenance

This exception is the sixth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

If a Hypervisor Maintenance exception exists and each attempt to execute an instruction when the Hypervisor Maintenance interrupt is enabled causes an exception (see the Programming Note below), the Hypervisor Maintenance interrupt is not delayed indefinitely.


These exceptions are the lowest priority exceptions. All have equal priority (i.e., the hardware may generate any one of the corresponding interrupts for which an exception exists). When one of

---

Programming Note

Some platform implementations may depend on timely servicing of Hypervisor Maintenance interrupts, e.g. to prevent physical damage. The Directed Ultravisor Doorbell interrupt handler may test the HMER to identify such circumstances and take appropriate action.
these exceptions is created, the interrupt processing mechanism waits for all other possible exceptions to be reported. It then generates the corresponding interrupt if no higher priority exception exists when the interrupt is to be generated.

If a Hypervisor Decrementer exception exists and each attempt to execute an instruction when the Hypervisor Decrementer interrupt is enabled causes an exception (see the Programming Note below), the Hypervisor Decrementer interrupt is not delayed indefinitely.

If LPES=0 and a Direct External exception exists and each attempt to execute an instruction when this interrupt is enabled causes an exception (see the Programming Note below), the Direct External interrupt is not delayed indefinitely.

7.10.3 EBB Classes
Event-based branches are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are "system-caused" are
- Performance Monitor
- External

Programming Note
An incorrect or malicious operating system could corrupt the first instruction in the interrupt vector location for an instruction-caused interrupt such that the attempt to execute the instruction causes the same exception that caused the interrupt (a looping interrupt; e.g., Trap instruction and Program interrupt). Similarly, the first instruction of the interrupt vector for one instruction-caused interrupt could cause a different instruction-caused interrupt, and the first instruction of the interrupt vector for the second instruction-caused interrupt could cause the first instruction-caused interrupt (e.g., Program interrupt and Floating-Point Unavailable interrupt). The looping caused by these and similar cases is terminated by the occurrence of a System Reset or Hypervisor Decrementer interrupt.

7.10 Relationship of Event-Based Branches to Interrupts

7.10.1 EBB Exception Priority
Event-based branches have a priority lower than that of all interrupts. When an event-based exception is created, the Event-Based Branch facility waits for all possible exceptions that would cause interrupts to be reported. It then generates the event-based branch if no exception that would cause an interrupt exists when the event-based branch is to be generated.

7.10.2 EBB Synchronization
When an event-based branch occurs, EBBRR is set to point to an instruction such that all preceding instruc-
Chapter 8. Timer Facilities

8.1 Overview

The Time Base, Decrementer, Hypervisor Decrementer, Processor Utilization of Resources, and Scaled Processor Utilization of Resources registers provide timing functions for the system. The remainder of this section describes these registers and related facilities.

8.2 Time Base (TB)

The Time Base (TB) is a 64-bit register (see Figure 69) containing a 64-bit unsigned integer that is incremented periodically.

```
   0  39
TBU40 ///
   0  32  63
TBU TBL
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBU40</td>
<td>Upper 40 bits of Time Base</td>
</tr>
<tr>
<td>TBU</td>
<td>Upper 32 bits of Time Base</td>
</tr>
<tr>
<td>TBL</td>
<td>Lower 32 bits of Time Base</td>
</tr>
</tbody>
</table>

Figure 69. Time Base

The Time Base is a hypervisor resource; see Chapter 2.

The SPRs TBU40, TBU, and TBL provide access to the fields of the Time Base shown in Figure 69. When a `mtspr` instruction is executed specifying one of these SPRs, the associated field of the Time Base is altered and the remaining bits of the Time Base are not affected.

See Chapter 5 of Book II for information about the update frequency of the Time Base.

The Time Base is implemented such that:

1. Loading a GPR from the Time Base has no effect on the accuracy of the Time Base.

2. Copying the contents of a GPR to the Time Base replaces the contents of the Time Base with the contents of the GPR.

The Power ISA does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock in a Power ISA system. The Time Base update frequency is not required to be constant. What is required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base is under the control of the system software.

Implementations must provide a means for either preventing the Time Base from incrementing or preventing it from being read in problem state (MSR_PP=1). If the means is under software control, it must be accessible only in hypervisor state (MSR_HVP=0b10). There must be a method for getting all Time Bases in the system to start incrementing with values that are identical or almost identical.
8.2.1 Writing the Time Base

Writing the Time Base is privileged, and can be done only in hypervisor state. Reading the Time Base is not privileged; it is discussed in Chapter 5 of Book II.

It is not possible to write the entire 64-bit Time Base using a single instruction. The `mttbl` and `mttbu` extended mnemonics write the lower and upper halves of the Time Base (TBL and TBU), respectively, preserving the other half. These are extended mnemonics for the `mtspr` instruction; Figure 19.

The Time Base can be written by a sequence such as:

```assembly
lwz Rx,upper # load 64-bit value for
lwz Ry,lower # TB into Rx and Ry
li Rz,0
mttbl Rz # set TBL to 0
mttbu Rx # set TBU
mttbl Ry # set TBU
```

Provided that no interrupts occur while the last three instructions are being executed, loading 0 into TBL prevents the possibility of a carry from TBL to TBU while the Time Base is being initialized.

The preferred method of changing the Time Base utilizes the TBU40 facility. The following code sequence demonstrates the process. Assume the upper 40 bits of Rx contain the desired value upper 40 bits of the Time Base.

```assembly
mftb Ry # Read 64-bit Time Base value
clrldi Ry,Ry,40 # lower 24 bits of old TB
mttbu40 Rx # write upper 40 bits of TB
mftb Rz # read TB value again
cmpld Rz,Ry # compare new and old lwr 24
bge done # no carry out of low 24 bits
addis Rx,Rx,0x0100 #increment upper 40 bits
mttbu40 Rx # update to adjust for carry
```

Programming Note

If software initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from $2^{64}-1$ to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

If Time Base bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0x0 only when bit 59 changes state regardless of whether or not they incremented to 0xF since they were previously set to 0x0.

See the description of the Time Base in Chapter 5 of Book II for ways to compute time of day in POSIX format from the Time Base.

8.3 Virtual Time Base

The Virtual Time Base (VTB) is a 64-bit incrementing counter.

![Figure 70. Virtual Time Base](#)

VTB

0 63

Virtual Time Base increments at the same rate as the Time Base until its value becomes 0xFFFFFFFF_0000_0000 (2^{64} - 1); at the next increment its value becomes 0x0000_0000_0000_0000. There is no interrupt or other indication when this occurs.

The operation of the Virtual Time Base has the following additional properties.

1. Loading a GPR from the Virtual Time Base has no effect on the accuracy of the Virtual Time Base.

2. Copying the contents of a GPR to the Virtual Time Base replaces the contents of the Virtual Time Base with the contents of the GPR.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Virtual Time Base input frequency will also change. Software must be aware of this in order to set interval timers.
8.4 Decrementer

The Decrementer (DEC) is a decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay.

The Decrementer is driven at the same frequency as the Time Base.

In configurations in which the hypervisor allows multiple partitions to time-share a processor, the Virtual Time Base can be managed by the hypervisor such that it appears to each partition as if it counts only during the times that the partition is executing.

In order to do this, the hypervisor saves the value of the Virtual Time Base as part of the program context when removing a partition from the processor, and restores it to its previous value when initiating the partition again on the same or another processor.

The LPCR is used to enable and disable Large Decrementer mode, as defined below. (See Section 2.2.)

When the Decrementer is not in Large Decrementer mode, it behaves as a 32-bit signed integer and operates as follows.

The Decrementer counts down until its value becomes 0x0000_0000_0000_0000; at the next decrement its value becomes 0x0000_0000_FFFF_FFFF. When reading the Decrementer using \texttt{mfspr}, bits 0:31 always read back as 0s.

When the contents of DEC\(_{32}\) change from 0 to 1, a Decrementer exception will come into existence within a reasonable period of time. When the contents of DEC\(_{32}\) change from 1 to 0, the existing Decrementer exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

The preceding paragraph applies regardless of whether the change in the contents of DEC\(_{32}\) is the result of decrementation of the Decrementer by the hardware or of modification of the Decrementer caused by execution of an \texttt{mtspr} instruction.

The Decrementer has the following additional properties.

1. Loading a GPR from the Decrementer has no effect on the accuracy of the Time Base.
2. Copying the contents of a GPR to the Decrementer replaces the contents of the Decrementer with the contents of the GPR.

When the Decrementer is in Large Decrementer mode, it behaves as a d-bit decrementing counter which is sign-extended to 64 bits. The value of d is implementation dependent but at least 32. When the Decrementer is written, bits 0:63-d are ignored by the hardware.

In Large Decrementer mode, the maximum positive value supported by the Decrementer is \(2^{d-1}-1\), represented with bits 0:64-d containing 0's and bits 65-d:63 containing 1's. The minimum value supported by the Decrementer is \(-2^{d-1}\), represented as 0xFFFF_FFFF_FFFF_FFFF.

When in Large Decrementer mode, the Decrementer operates as follows.

The binary value of the Decrementer counts down until its value becomes 0x0000_0000_0000_0000; at the next decrement its value becomes the minimum value supported, which is represented as 0xFFFF_FFFF_FFFF_FFFF.

The contents of the DEC\(_{0}\) change from 0 to 1, a Decrementer exception will come into existence within a reasonable period of time. When the contents of DEC\(_{0}\) change from 1 to 0, the existing Decrementer exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

The preceding paragraph applies regardless of whether the change in the contents of DEC\(_{0}\) is the result of decrementation of the Decrementer by the hardware or of modification of the Decrementer caused by execution of an \texttt{mtspr} instruction.

In systems that change the Time Base update frequency for purposes such as power management, the Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

If Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.
8.4.1 Writing and Reading the Decrementer

The contents of the Decrementer can be read or written using the `mfspr` and `mtspr` instructions, both of which are privileged when they refer to the Decrementer. Using an extended mnemonic (Figure 19), the Decrementer can be written from GPR Rx using:

```
mtdec Rx
```

The Decrementer can be read into GPR Rx using:

```
mfdec Rx
```

Copying the Decrementer to a GPR has no effect on the Decrementer contents or on the interrupt mechanism.

8.5 Hypervisor Decrementer

The Hypervisor Decrementer is a h-bit decrementing counter that is sign-extended to 64 bits. The value of h is implementation dependent, however the number of bits supported by the Hypervisor Decrementer must be greater than or equal to the number of bits supported by the Decrementer. When the Decrementer is written, bits 0:63-h are ignored by the hardware.

The binary value of the Hypervisor Decrementer counts down until its value becomes 0x0000_0000_0000_0000; at the next decrement its value becomes the minimum value supported, which is represented as 0xFFFF_FFFF_FFFF_FFFF.

When the contents of HDEC0 change from 0 to 1 and the thread is not in a power-saving mode, a Hypervisor Decrementer exception will come into existence within a reasonable period of time. When a Hypervisor Decrementer interrupt occurs, the existing Hypervisor Decrementer exception will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event. Even if multiple HDEC0 change transitions from 0 to 1 occur before a Hypervisor Decrementer interrupt occurs, at most one Hypervisor Decrementer exception exists.

The preceding paragraph applies regardless of whether the change in the contents of HDEC0 is the result of decrementation of the Hypervisor Decrementer by the hardware or of modification of the Hypervisor Decrementer caused by execution of an `mtspr` instruction.

The operation of the Hypervisor Decrementer has the following additional properties.

1. Loading a GPR from the Hypervisor Decrementer has no effect on the accuracy of the Hypervisor Decrementer.
2. Copying the contents of a GPR to the Hypervisor Decrementer replaces the contents of the Hypervisor Decrementer with the contents of the GPR.

### Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Hypervisor Decrementer update frequency will also change. Software must be aware of this in order to set interval timers.

If Hypervisor Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.

### Programming Note

A Hypervisor Decrementer exception is not created if the thread is in a power-saving mode when HDEC0 changes from 0 to 1 because having a Hypervisor Decrementer interrupt occur almost immediately after exiting the power-saving mode in this case is deemed unnecessary. The hypervisor already has control, and if a timed exit from the power-saving mode is necessary and possible, the hypervisor can use the Decrementer to exit the power-saving mode at the appropriate time. For some power-saving levels, the state of the Hypervisor Decrementer and Decrementer is not necessarily maintained and updated.

8.6 Processor Utilization of Resources Register (PURR)

The Processor Utilization of Resources Register (PURR) is a 64-bit counter, the contents of which provide an estimate of the resources used by the thread. The contents of the PURR are treated as a 64-bit unsigned integer.

```
<table>
<thead>
<tr>
<th>PURR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

**Figure 72. Processor Utilization of Resources Register**

The PURR is a hypervisor resource; see Chapter 2.
The contents of the PURR increase monotonically, unless altered by software, until the sum of the contents plus the amount by which it is to be increased exceed $0xFFFF_FFFF_FFFF_FFFF$ ($2^{64} - 1$) at which point the contents are replaced by that sum modulo $2^{64}$. There is no interrupt or other indication when this occurs.

The rate at which the value represented by the contents of the PURR increases is an estimate of the portion of resources used by the thread per unit time with respect to other threads that share those resources monitored by the PURR. When the thread is idle, the rate at which the PURR value increases is implementation dependent.

Let the difference between the value represented by the contents of the Time Base at times $T_a$ and $T_b$ be $T_{ab}$. Let the difference between the value represented by the contents of the PURR at time $T_a$ and $T_b$ be the value $P_{ab}$. The ratio of $P_{ab}/T_{ab}$ is an estimate of the percentage of shared resources used by the thread during the interval $T_{ab}$. For the set ($S$) of threads that share the resources monitored by the PURR, the sum of the usage estimates for all the threads in the set is 1.0.

The definition of the set of threads $S$, the shared resources corresponding to the set $S$, and specifics of the algorithm for incrementing the PURR are implementation-specific.

The PURR is implemented such that:

1. Loading a GPR from the PURR has no effect on the accuracy of the PURR.
2. Copying the contents of a GPR to the PURR replaces the contents of the PURR with the contents of the GPR.

### Programming Note

Estimates computed as described above may be useful for purposes related to resource utilization, including utilization-based system management and planning.

Because the rate at which the PURR accumulates resource usage estimates is dependent on the frequency at which the Time Base is incremented, and the frequency of the oscillator that drives instruction execution may vary independently from that of the Time Base, the interpretation of the contents of the PURR may be inaccurate as a measurement of capacity consumption for accounting purposes. The SPURR should be used for accounting purposes.

### 8.7 Scaled Processor Utilization of Resources Register (SPURR)

The Scaled Processor Utilization of Resources Register (SPURR) is a 64-bit counter, the contents of which provide an estimate of the resources used by the thread. The contents of the SPURR are treated as a 64-bit unsigned integer.

The SPURR is a hypervisor resource; see Section 2.6.

The contents of the SPURR increase monotonically, unless altered by software, until the sum of the contents plus the amount by which it is to be increased exceed $0xFFFF_FFFF_FFFF_FFFF$ ($2^{64} - 1$) at which point the contents are replaced by that sum modulo $2^{64}$. There is no interrupt or other indication when this occurs.

The rate at which the value represented by the contents of the SPURR increases is an estimate of the portion of resources used by the thread with respect to other threads that share those resources monitored by the SPURR, and relative to the computational capacity provided by those resources. The computational capacity provided by the shared resources may vary as a function of the frequency of the oscillator which drives the resources or as a result of deliberate delays in processing that are created to reduce power consumption. When the thread is idle, the rate at which the SPURR value increases is implementation dependent.

Let the difference between the value represented by the contents of the Time Base at times $T_a$ and $T_b$ be $T_{ab}$. Let the ratio of the effective and nominal frequencies of the oscillator driving instruction execution $f_e/f_n$ be $f_r$. Let the ratio of delay cycles created by power reduction circuitry and total cycles $c_d/c_t$ be $c_r$. Let the difference between the value represented by the contents of the SPURR at time $T_a$ and $T_b$ be the value $S_{ab}$. The ratio of $S_{ab}/(T_{ab} x f_r x (1 - c_r))$ is an estimate of the percentage of shared resource capacity used by the thread during the interval $T_{ab}$. For the set ($S$) of threads that share the resources monitored by the SPURR, the sum of the usage estimates for all the threads in the set is 1.0.

The definition of the set of threads $S$, the shared resources corresponding to the set $S$, and specifics of the algorithm for incrementing the SPURR are implementation-specific.

The SPURR is implemented such that:

1. Loading a GPR from the SPURR has no effect on the accuracy of the SPURR.
2. Copying the contents of a GPR to the SPURR replaces the contents of the SPURR with the contents of the GPR.

Programming Note
Estimates computed as described above may be useful for purposes of resource use accounting, program dispatching, etc.

8.8 Instruction Counter

The Instruction Counter (IC) is a 64-bit incrementing counter that counts the number of instructions that the thread has completed (according to the sequential execution model; see Section 2.2 of Book I).

Figure 74. Instruction Counter
Chapter 9. Debug Facilities

9.1 Overview

Implementations provide debug facilities to enable hardware and software debug functions, such as control flow tracing, data address watchpoints, and program single-stepping. The debug facilities described in this section consist of the Come-From Address Register (see Section 9.2), Completed Instruction Address Breakpoint Register (see Section 9.3), and the Data Address Watchpoint Register (DAWRn) and Data Address Watchpoint Register Extension (DAWRXn) (see Section 9.4). The interrupt associated with the Data Address Breakpoint registers is described in Section 7.5.3. The interrupt associated with the Completed Instruction Address Breakpoint Register is described in Section 7.5.15. The Trace facility, which can be used for single-stepping as well as for control flow tracing, is described in Section 7.5.15.

The \texttt{mfspr} and \texttt{mtspr} instructions (see Section 5.4.4) provide access to the registers of the debug facilities.

In addition to the facilities mentioned above, implementations typically provide debug facilities, modes, and access mechanisms that are implementation-specific. For example, implementations typically provide facilities for instruction address tracing, and also access to certain debug facilities via a dedicated interface such as the IEEE 1149.1 Test Access Port (JTAG).

9.2 Come-From Address Register

The Come-From Address Register (CFAR) is a 64-bit register. When an \texttt{rfebb}, \texttt{rfid}, or \texttt{rfscv} instruction is executed, the register is set to the effective address of the instruction. When a \texttt{Branch} instruction is executed and the branch is taken, the register is set to the effective address of an instruction in the instruction cache block containing the \texttt{Branch} instruction, except that if the \texttt{Branch} instruction is a B-form \texttt{Branch} (i.e., \texttt{bc}, \texttt{bca}, \texttt{bcl}, or \texttt{bcla}) for which the target address is in the instruction cache block containing the \texttt{Branch} instruction or is in the previous or next cache block, the register is not necessarily set. For \texttt{Branch} instructions, the setting need not occur until a subsequent context synchronizing operation has occurred.

![Figure 75. Come-From Address Register](image)

The contents of the CFAR can be read and written using the \texttt{mfspr} and \texttt{mtspr} instructions. Access to the CFAR is privileged.

--- Programming Note ---

This register can be used for purposes of debugging software. For example, often a software bug results in the program executing a portion of the code that it should not have reached or causing an unexpected interrupt. In the former case, a breakpoint can be placed in the portion of the code that was erroneously reached and the program reexecuted. In either case, the interrupt handler can save the contents of the CFAR (before executing the first instruction that would modify the register), and then make the saved contents available for a debugger to use in determining the control flow path by which the exception was reached.

In order to preserve the CFAR's contents for each partition and to prevent it from being used to implement a "covert channel" between partitions, the hypervisor should initialize/save/restore the CFAR when switching partitions on a given thread.

9.3 Completed Instruction Address Breakpoint

The Completed Instruction Address Breakpoint mechanism provides a means of detecting an instruction completion at a specific instruction address. The address comparison is done on an effective address (EA).

The Completed Instruction Address Breakpoint mechanism is controlled by the Completed Instruction Address Breakpoint Register (CIABR), shown in Figure 76, except that if SMFCTRLD=1 when PRIV≠0,
the Privilege specification in the PRIV field is ignored and the facility detects instruction address matches in ultravisor state.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:61</td>
<td>CIEA</td>
<td>Completed Instruction Effective Address</td>
</tr>
<tr>
<td>62:63</td>
<td>PRIV</td>
<td>Privilege (PRIV &gt; 0b00 ignored when SMFCTRLD=1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00: Disable matching</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01: Match in problem state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10: Match in privileged non-hypervisor state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11: Match in hypervisor non-ultravisor state</td>
</tr>
</tbody>
</table>

Figure 76. Completed Instruction Address Breakpoint Register

A Completed Instruction Address Breakpoint match occurs upon instruction completion if all of the following conditions are satisfied. The values of CIABR, SMFCTRL, and the MSR that are used for the comparisons are those that exist at the time the instruction is initiated.

- the completed instruction address is equal to CIEA_{0:61} \| 0b00. For prefixed instructions, the completed instruction address is the address of the prefix.
- SMFCTRLD=0 and the thread privilege matches that specified in PRIV or SMFCTRLD=1, PRIV ≠ 0, and MSR_{HV PR}=0b110.

In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

A Completed Instruction Address Breakpoint match causes a Trace exception, which may cause a Trace interrupt as described in Section 7.5.15.

9.4 Data Address Watchpoint

The Data Address Watchpoint mechanism provides a means of detecting load and store accesses to multiple doubleword-aligned effective address (EA) ranges. At least two independent address ranges are provided.

Programming Note

The Data Address Watchpoint mechanism employs a simple EA compare. It makes no attempt to take the radix table translation quadrants (keyed off EA_{0:1}) into account to enable a single setting to work in all privilege levels.

Each Data Address Watchpoint range is controlled by a pair of SPRs: the Data Address Watchpoint Register (DAWRn), shown in Figure 77, and the Data Address Watchpoint Register Extension (DAWRXn), shown in Figure 78, where n=0,1, ..., SMFCTRLD functions as an extension to the PRIVM field: when SMFCTRLD=1, the facility detects data address watchpoint matches in ultravisor state in addition to states enabled by the PRIVM field.

Figure 77. Completed Instruction Address Breakpoint Register

A Completed Instruction Address Breakpoint match occurs upon instruction completion if all of the following conditions are satisfied. The values of CIABR, SMFCTRL, and the MSR that are used for the comparisons are those that exist at the time the instruction is initiated.

- the completed instruction address is equal to CIEA_{0:61} \| 0b00. For prefixed instructions, the completed instruction address is the address of the prefix.
- SMFCTRLD=0 and the thread privilege matches that specified in PRIV or SMFCTRLD=1, PRIV ≠ 0, and MSR_{HV PR}=0b110.

In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

A Completed Instruction Address Breakpoint match causes a Trace exception, which may cause a Trace interrupt as described in Section 7.5.15.

9.4 Data Address Watchpoint

The Data Address Watchpoint mechanism provides a means of detecting load and store accesses to multiple doubleword-aligned effective address (EA) ranges. At least two independent address ranges are provided.

Programming Note

The Data Address Watchpoint mechanism employs a simple EA compare. It makes no attempt to take the radix table translation quadrants (keyed off EA_{0:1}) into account to enable a single setting to work in all privilege levels.

Each Data Address Watchpoint range is controlled by a pair of SPRs: the Data Address Watchpoint Register (DAWRn), shown in Figure 77, and the Data Address Watchpoint Register Extension (DAWRXn), shown in Figure 78, where n=0,1, ..., SMFCTRLD functions as an extension to the PRIVM field: when SMFCTRLD=1, the facility detects data address watchpoint matches in ultravisor state in addition to states enabled by the PRIVM field.

Figure 77. Data Address Watchpoint Register

Figure 78. Data Address Watchpoint Register Extension

The supported PRIVM values are 0b000, 0b001, 0b010, 0b011, 0b100, and 0b111 when SMFCTRLD=0 and 0b000, 0b001, 0b010, and 0b011 when SMFCTRLD=1. If the combination of SMFCTRLD and the PRIVM field does not contain one of the supported values, then whether a match occurs for a given storage access is undefined. Elsewhere in this section it is assumed that the PRIVM field contains one of the supported values.
When SMFCTRLD=0, PRIMV value 0b000 causes matches not to occur regardless of the contents of other DAWRx and DAWRxP fields. PRIMV values 0b101 and 0b110 are not supported because a storage location that is shared between the hypervisor and non-hypervisor software is unlikely to be accessed using the same EA by both the hypervisor and the non-hypervisor software. (PRIMV value 0b111 is supported primarily for reasons of software compatibility with respect to emulation of the DABR facility as described in a subsequent Programming Note.)

SMFCTRLD=1 is provided for ultravisor debugging and also for ultravisor supervision of secure partition debugging. When SMFCTRLD=1, exceptions due to matches that occur in hypervisor non-ultravisor state are unlikely to be desirable.

A Data Address Watchpoint match occurs for a Load or Store instruction, or for an instruction that is treated as a Load or Store, if, for any byte accessed, all of the following conditions are satisfied. For the first condition, chk_DEAW and chk_EA are defined as follows. If MSR[HV DB] = 0b10 and HRAMMC=1 then

chk_DEAW = 0b0 || DEAW1:60 and
chk_EA = 0b0 || EA1:63;

otherwise
chk_DEAW = DEAW and
chk_EA = EA.

The access is located in the range
chk_DEAW0:60 ≤ chk_EA0:60 ≤ (chk_DEAW0:60 + (550 || MRD0:5)) such that
(chk_EA0:60 AND (551 || 60)) = (chk_DEAW0:60 AND (551 || 60)).

(MSR[DR] = DAWRXnWT) | DAWRXnWT1

the thread is in
- ultravisor state and SMFCTRLD=1, or
- hypervisor non-ultravisor state and
DAWRXnhyp = 1, or
- privileged non-hypervisor state and
DAWRXnpnh = 1, or
- problem state and DAWRXnp = 1

the instruction is a Store or treated as a Store and
DAWRXnDW = 1, or the instruction is a Load or treated as a Load and DAWRXnDR = 1.

In hypervisor and ultravisor real addressing modes, bits 1:63-m of the EA are ignored for the purpose of detecting a match, where m is the real address size supported by the implementation. In virtual real addressing mode, the high order 24 bits of the EA are ignored for the purpose of detecting a match. In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

A watched range must not cross any of the following boundaries in their respective circumstances. If it does, the facility will operate correctly with respect to the various control parameters (e.g., PRIMV), but the set of EAs that cause matches is undefined.

- the 2^32-byte boundary when HPT translation is being performed in other than virtual real addressing mode (i.e., a range that includes the last and first byte of the 2^32-byte effective address space)
- a 2^32-byte boundary when the thread is in hypervisor or ultravisor real addressing mode (i.e., a range that, if the corresponding EAs were used to address storage on a design that ignores the high-order bits of the 60-bit real address that are not supported by the implementation, would include the last and first byte of the 2^32-byte effective address space)
- a 2^32-byte boundary when Radix Tree translation is being performed (i.e., a quadrant boundary)
- a 2^32-byte boundary when HPT translation is being performed in virtual real addressing mode (i.e., a range that, if the corresponding EAs were used to address storage, would include the last and first byte of the reserved virtual segment)
- a 2^32-byte boundary when the thread is in 32-bit mode (i.e., a range that, if the corresponding EAs were used to address storage, would include the last and first byte of the 2^32-byte effective address space)

If the above conditions are satisfied, it is undefined whether a match occurs in the following cases.

- The instruction is Store Conditional but the store is not performed
- The instruction is dcbz. (For the purpose of determining whether a match occurs, dcbz is treated as a Store.)

The Cache Management instructions other than dcbz never cause a match.

A Data Address Watchpoint match causes a Data Storage exception or a Hypervisor Data Storage exception (see Section 7.5.3, “Data Storage Interrupt (DSI)” on page 1266 and Section 7.5.16, “Hypervisor Data Storage Interrupt (HDSI)” on page 1277). If a match occurs, some or all of the bytes of the storage operand may have been accessed; however, if a Store instruction causes the match, the storage operand is not modified if the instruction is one of the following:

- any Store instruction that causes an atomic access

Programming Note

The Data Address Watchpoint mechanism does not apply to instruction fetches.
The limitations of the Data Address Watchpoint mechanism may lead to uncertainties when attempting to interpret a reported match. For example, there is no explicit indication of which watchpoint matched. The watched address ranges could overlap, for example, as a result of the doubleword granularity of the address range specification, making it unclear which watchpoint matched. Even if one of the overlapping watchpoints was set to match only for stores and the other only for loads, the existence of the Atomic Memory Operations would make the reporting of a store in the DSISR inconclusive. For a given watchpoint, the doubleword granularity may make the validity of the match uncertain because only the starting address of the match is reported without a length. In such cases, it is necessary to examine the operand that caused the match to determine what actually happened.

Implementations that comply with versions of the architecture that precede Version 2.02 do not provide the DABRX (now replaced by DAWRXn). Forward compatibility for software that was written for such implementations (and uses the Data Address Breakpoint facility) can be obtained by setting DAWRXn[60:63] to 0b0111.
Chapter 10. Performance Monitor Facility

10.1 Overview

The Performance Monitor facility provides a means of collecting information about program and system performance.

10.2 Performance Monitor Operation

The Performance Monitor facility includes the following features.

- an MSR bit
  - PMM (Performance Monitor Mark), which can be used to select one or more programs for monitoring

- registers
  - PMC1 - PMC6 (Performance Monitor Counters 1 - 6), which count events
  - MMCR0, MMCR1, MMCR2, MMCR3 and MMCRA (Monitor Mode Control Registers 0, 1, 2, 3 and A), which control the Performance Monitor facility
  - SIAR, SDAR, SIER, SIER2 and SIER3 (Sampled Instruction Address Register, Sampled Data Address Register, Sampled Instruction Event Register, Sampled Instruction Event Register 2 and Sampled Instruction Event Register 3), which contain the address of the “sampled instruction” and of the “sampled data,” and additional information about the “sampled instruction” (see Section 10.4.8 - Section 10.4.10)

- the Performance Monitor interrupt and Performance Monitor event-based branch, which can be caused by monitored conditions and events.

A “counter negative condition” exists when the value in a PMC is negative (i.e., when bit 0 of the PMC is 1). A “Time Base transition event” occurs when a selected bit of the Time Base changes from 0 to 1 (the bit is selected by a field in MMCR0). The term “condition or event” is used as an abbreviation for “counter negative condition or Time Base transition event”. A condition or event can be caused implicitly by the hardware (e.g., incrementing a PMC) or explicitly by software (mtspr).

A condition or event is enabled if the corresponding “Enable” bit (i.e., PMC1CE, PMCjCE, or TBEE) in MMCR0 is 1. The occurrence of an enabled condition or event can have side effects within the Performance Monitor, such as causing the PMCs to cease counting.

An enabled condition or event causes a Performance Monitor alert if Performance Monitor alerts are enabled by the corresponding “Enable” bit in MMCR0. Another cause of a Performance Monitor alert is the threshold event counter reaching its maximum value (see Section 10.4.3). A single Performance Monitor alert may reflect multiple enabled conditions and events.

When a Performance Monitor alert occurs, MMCROPMAC is set to 1 and the writing of BHRB entries, if in process, is suspended.

When the contents of MMCROPMAC change from 0 to 1, a Performance Monitor exception will come into existence within a reasonable period of time. When the contents of MMCROPMAC change from 1 to 0, the existing Performance Monitor exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

A Performance Monitor exception causes one of the following.

- If MSR_EE = 1, MMCRO_EBE = 0, and either HFSCR_PM=1 or the thread is in hypervisor state, an interrupt occurs.
- If MSR_PR = 1, MMCRO_EBE = 1, a Performance Monitor event-based exception occurs if

Many aspects of the operation of the Performance Monitor are summarized by the following hierarchy, which is described starting at the lowest level.
BESCR\textsubscript{PME}=1, provided that event-based exceptions are enabled by FSCRE\textsubscript{EBB} and HFSCRE\textsubscript{EBB}. When a Performance Monitor event-based exception occurs, an event-based branch is generated if BESCR\textsubscript{OE}=1.

**Programming Note**

The Performance Monitor can be effectively disabled (i.e., put into a state in which Performance Monitor SPRs are not altered and Performance Monitor exceptions do not occur) by setting MMCR0 to 0x0000_0000_8000_0000.

The Performance Monitor also controls when BHRB entries are written, the instruction filters that are used when writing BHRB entries, and the availability of the BHRB in problem state. It also controls whether Performance Monitor exceptions cause Performance Monitor event-based exceptions or Performance Monitor interrupts. See Section 10.4.4.

### 10.3 No-op Instructions

**Reserved for the Performance Monitor**

The following forms of the \texttt{and x,x,x} instruction are reserved for exclusive use by the Performance Monitor.

- \texttt{and x,x,x}, where x=0,1.

**Programming Note**

An example usage of a probe no-op by the Performance Monitor is to measure branch prediction effectiveness. In order to do this, one of probe no-ops is inserted in various sections of the code in which branch prediction efficiency is being studied. The Performance Monitor registers are then set up as follows.

**MMCRA:**

- ES=010 (only probe no-ops eligible for sampling)
- SM=00 (all eligible instructions)
- SE=1 (enable random sampling).

Other fields in MMCRA are set as desired.

**MMCRI:**

- PMC1SEL=E0 (count PMC1 on dispatch)
- PMC4SEL=E0 (count PMC4 on completion)

Other counters initialized as desired.

**MMCRC:**

- Initialize as desired.

**MMCR0:**

- FC is set to 0 to stop freezing the counters
- PMAE is set to 1 to enable PMU alerts.

Other fields in MMCR0 are set as desired.

Subsequently, when a PMU alert occurs, PMCs 1 and 4 can be read. The difference between the two counter values provides an indication of branch prediction effectiveness in the areas of the code in which the probe no-op was inserted.

### 10.4 Performance Monitor Facility Registers

The Performance Monitor registers count events, control the operation of the Performance Monitor, and provide associated information.

The elapsed time between the execution of an instruction and the time at which events due to that instruction have been reflected in Performance Monitor registers is not defined. No means are provided by which software can ensure that all events due to preceding instructions have been reflected in Performance Monitor registers. Similarly, if the events being monitored may be caused by operations that are performed out-of-order, no means are provided by which software can prevent such events due to subsequent instructions from being reflected in Performance Monitor registers. Thus the contents obtained by reading a Performance Monitor register may not be precise: it may fail to reflect some events due to instructions that precede the \texttt{mfspkr} and may reflect some events due to instructions that follow the \texttt{mfspkr}. This lack of precision applies regardless of whether the state of the thread is such that the register is subject to change by the hardware at the time the
**10.4.1 Performance Monitor SPR Numbers**

The Performance Monitor registers have two sets of SPR numbers, one set that is non-privileged and another set that is privileged.

For the purpose of explanation elsewhere in the architecture, the non-privileged registers are divided into two groups as defined below.

- **A**: The non-privileged read/write Performance Monitor registers (i.e., the PMCs, MMCR0, MMCR2, and MMCRA at SPR numbers 771-776, 779, 769, and 770, respectively).

- **B**: The non-privileged read-only Performance Monitor registers (i.e., SIER2, SIER3, MMCR3, SIER, SIAR, SDAR, and MMCR1 at SPR numbers 736, 737, 738, 768, 780, 781, and 782, respectively).

The SPRs in group B are treated as undefined registers for write (mtspr) operations. See the mtspr instruction description in Section 5.4.4 for additional information.

When the PCR makes a register in either group A or B unavailable in problem state, that SPR is not included in group A or B.

**Programming Note**

Older versions of Performance Monitor facilities used different sets of SPR numbers from those shown in Section 5.4.4. (All 32-bit PowerPC implementations used a different set.)

**10.4.2 Performance Monitor Counters**

The six Performance Monitor Counters, PMC1 through PMC6, are 32-bit registers that count events.

**Programming Note**

Depending on the events being monitored, the contents of Performance Monitor registers may be affected by aspects of the runtime environment (e.g., cache contents) that are not directly attributable to the programs being monitored.

**Programming Note**

PMC5 and PMC6 are defined to facilitate calculating basic performance metrics such as cycles per instruction (CPI).

**Programming Note**

Software can use a PMC to "pace" the collection of Performance Monitor data. For example, if it is desired to collect event counts every n cycles, software can specify that a particular PMC count cycles, and set that PMC to 0x8000_0000 - n. The events of interest would be counted in other PMCs. The counter negative condition that will occur after n cycles can, with the appropriate setting of MMCR bits, cause counter values to become frozen, cause a Performance Monitor exception to occur, etc.

**10.4.2.1 Event Counting and Sampling**

The PMCs are enabled to count unless they are “frozen” by one or more of the “freeze counters” fields in MMCR0 or MMCR2.

Each of PMC’s 1-4 can be configured, using MMCR1, to count “continuous” events (events that can occur at any time), or to count “randomly sampled” events (or “sampled” events) that are associated with the execution of randomly sampled instructions.
Continuous events always cause the counters to count (unless counters are frozen). These events are specified for each counter by using encodes F0-FF in the PMCn Selector fields in MMCRI.

Randomly sampled events can cause the counters to count only when random sampling has been enabled by setting MMCRA[SE]=1. The types of instructions that are sampled are specified in MMCRA[SM] and MMCRA[ES]. Randomly sampled events are specified for each counter by using encodes E0-EF in the PMCn Selector fields in MMCRI.

**Programming Note**

A typical sequence of operations that enables use the PMCs is as follows.

- Freeze the counters by setting MMCR0[FC]=1.
- Set control fields in MMCR0 and MMCR2 that control counting in various privilege states and other modes, and that enable counter negative conditions.
- Initialize the events to be counted by PMCs 1-4 using the PMCn Selector fields in MMCRI.
- Specify the BHRB filtering mode, threshold event Counter events, and whether or not random sampling is enabled in the corresponding fields in MMCRA.
- Initialize the PMCs to the values desired. For example, in order to configure a counter to cause a counter negative condition after n counts, that counter would be initialized to $2^{32-n}$.
- Set MMCR0[FC] to 0 to disable freezing the counters, and set MMCR0[PMAE] to 1 if a Performance Monitor alert (and the corresponding Performance Monitor interrupt) is desired when an enabled condition or event occurs. (See Section 10.2 for the definition of enabled condition or event.)

When the Performance Monitor alert occurs, the program would typically read the values of the counters as well as the contents of SIAR, SDAR, SIER as needed in order to extract the information that was being monitored.

See Sections 10.4.4 - 10.4.10 for information regarding MMCRs, SIAR, SDAR, and SIER, and some additional usage examples.

### 10.4.3 Threshold Event Counter

The threshold event counter and associated controls are in MMCRA (see Section 10.4.7). When Performance Monitor alerts are enabled (MMCR0[PMAE]=1), this counter begins incrementing from value 0 upon each occurrence of the event specified in the Threshold Event Counter Event (TECE) field after the event specified by the Threshold Start Event (TS) field occurs. The counter stops incrementing when the event specified in the Threshold End Event (TE) field occurs. The counter subsequently freezes until the event specified in the TS field is again recognized, at which point it restarts incrementing from value 0 as explained above. If the counter reaches its maximum value or a Performance Monitor alert occurs, incrementing stops. After the Performance Monitor alert occurs, the contents of the threshold event counter are not altered by the hardware until software sets MMCR0[PMAE] to 1.

**Programming Note**

Because hardware can modify the contents of the threshold event counter when random sampling is enabled (MMCRA[SE]=1) and MMCR0[PMAE]=1 at any time, any value written to the threshold event counter under this condition may be immediately overwritten by hardware.

The threshold event counter value is represented as a 3-bit integral power of 4, multiplied by a 7-bit integer. The exponent is contained in MMCRATECX, and the multiplier is contained in MMCRA[TECM]. For a given counter exponent, e, and multiplier, m, the number represented is as follows:

$$ N = 4^e \times m $$

This counter format allows the counter to represent a range of 0 through approximately 2 million counts with many fewer bits than would be required by a binary counter.

To represent a given counter value, hardware uses as e the smallest 3-bit integer for which a 7-bit integer exists such that the given counter value can be expressed using this format.

**Programming Note**

Software can obtain the number N from the contents of the threshold event counter by shifting the multiplier left twice times the value contained in the exponent.

The value in the counter is the exact number of events that occur for values from 0 through the maximum multiplier value (127), with 4 events of the exact value for values from 128 - 508 (or 127×4), within 16 events of the exact value for values from 512 - 2032 (or 127×4²), and so on. This represents an event count accuracy of approximately 3%, which is expected to be sufficient for most situations in which a count of events between a start and end event is required.
10.4.4 Monitor Mode Control Register 0

Monitor Mode Control Register 0 (MMCR0) is a 64-bit register as shown below.

<table>
<thead>
<tr>
<th>MMCR0</th>
<th>0</th>
<th>31</th>
</tr>
</thead>
</table>

Figure 80. Monitor Mode Control Register 0

MMCR0 is used to control multiple functions of the Performance Monitor. Some fields of MMCR0 are altered by the hardware when various events occur.

The following notation is used in the definitions below. “PMCs” refers to PMCs 1 - n and “PMCj” refers to PMCj, where 2 ≤ j ≤ n. n=4 when MMCR0PMCC=0b11 and n=6 otherwise.

When MMCR0PMCC is set to 0b10 or 0b11, providing problem state programs read/write access to MMCR0, only FC, PMAE, PMAO can be accessed. All other bits are not changed when mtspr is executed in problem state, and all other bits return 0s when mfspr is executed in problem state.

Programming Note

When using the threshold event counter, software typically specifies a “threshold counter exceeded n” event in MMCR1. This enables a PMC to count the number of times the counter exceeded a specified threshold value during the time Performance Monitor alerts were enabled.

The bit definitions of MMCR0 are as follows.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Reserved</td>
<td>0:31</td>
</tr>
<tr>
<td>32</td>
<td>Freeze Counters (FC)</td>
<td>0: The PMCs are incremented (if permitted by other MMCR bits). 1: The PMCs are not incremented.</td>
</tr>
<tr>
<td></td>
<td>The hardware sets this bit to 1 when an enabled condition or event occurs and MMCR0FCECE=1.</td>
<td></td>
</tr>
</tbody>
</table>

Programming Note

In order to freeze counters in problem state regardless of MSRHV, MMCR0FCPC must be set to 0 and MMCR0FCP must be set to 1.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>Freeze Counters in Privileged State (FCS)</td>
<td>0: The PMCs are incremented (if permitted by other MMCR bits). 1: The PMCs are not incremented if MSRHVPR=0b00.</td>
</tr>
<tr>
<td></td>
<td>Conditionally Freeze Counters and BHRB in Problem State (FCP)</td>
<td>If the value of bit 51 (FCPC) is 0, this field has the following meaning. 0: The PMCs are incremented (if permitted by other MMCR bits) and entries are written into the BHRB (if permitted by the BHRB Instruction Filtering Mode field in MMCR0). 1: The PMCs are not incremented, and entries are not written into the BHRB, if MSRPR=1.</td>
</tr>
<tr>
<td>34</td>
<td>If the value of bit 51 (FCPC) is 1, this field has the following meaning. 0: The PMCs are not incremented, and entries are not written into the BHRB, if MSRHVPR=0b01. 1: The PMCs are not incremented, and entries are not written into the BHRB, if MSRHVPR=0b11.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>Freeze Counters while Mark = 1 (FCM1)</td>
<td>0: The PMCs are incremented (if permitted by other MMCR bits). 1: The PMCs are not incremented if MSRPM=1.</td>
</tr>
<tr>
<td></td>
<td>Freeze Counters while Mark = 0 (FCM0)</td>
<td>0: The PMCs are incremented (if permitted by other MMCR bits). 1: The PMCs are not incremented if MSRPM=0.</td>
</tr>
</tbody>
</table>
| 36     | Performance Monitor Alert Enable (PMAE) | 0: Performance Monitor alerts are disabled and BHRB entries are not written. 1: Performance Monitor alerts are enabled, and BHRB entries are written (if enabled by other bits) until a Performance Monitor alert occurs, at which time:  
- MMCR0PMAE is set to 0  
- MMCR0PMAO is set to 1  
| 37     | | |
Freeze Counters on Enabled Condition or Event (FCECE)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are incremented (if permitted by other MMCR bits) until an enabled condition or event occurs when MMCR0_INSTANCE=0, at which time:
   - MMCR0_FC is set to 1

If the enabled condition or event occurs when MMCR0_INSTANCE=1, the FCECE bit is treated as if it were 0.

Time Base Selector (TBSEL)

This field selects the Time Base bit that can cause a Time Base transition event (the event occurs when the selected bit changes from 0 to 1).

00 Time Base bit 47 is selected.
01 Time Base bit 51 is selected.
10 Time Base bit 55 is selected.
11 Time Base bit 63 is selected.

Time Base Event Enable (TBEE)

0 Time Base transition events are disabled.
1 Time Base transition events are enabled.

BHRB Available (BHRBA)

This field controls whether the BHRB instructions are available in problem state. If an attempt is made to execute a BHRB instruction in problem state when the BHRB instructions are not available, a Facility Unavailable interrupt will occur.

0 clrbhrb and mfbhrbe are not available in problem state.
1 clrbhrb and mfbhrbe are available in problem state unless they have been made unavailable by some other register.

Performance Monitor Event-Based Branch Enable (EBE)

This field controls whether Performance Monitor event-based branches and Performance Monitor event-based exceptions are enabled.

When Performance Monitor event-based branches and exceptions are disabled, no Performance Monitor event-based branches or exceptions occur regardless of the state of BESCR_PME.
0  Performance Monitor event-based branches and exceptions are disabled.
1  Performance Monitor event-based branches and exceptions are enabled.

**Programming Note**

In order to enable a problem state application to use the event-based Branch facility for Performance Monitor events, privileged software initializes MMCR1 to specify the events to be counted, and sets MMCR2, and MMCRA to specify additional sampling controls. MMCR0 should be initialized with PMCC set to 0b10 or 0b11 (to give problem state access to various Performance Monitor registers), PMAE and PMAO set to 0s (disabling Performance Monitor alerts), and EBE set to 1 (enabling Performance Monitor event-based branches and exceptions to occur). If the Event-Based Branch facility has not been enabled in the FSCR and HFSCR, it must be enabled in these registers as well.

The above operations by the operating system enable the application to control Performance Monitor event-based branching by means of BESCRPME (to enable or disable Performance Monitor event-based branching) and MMCR0PMAE (to enable or disable Performance Monitor alerts).

**PMC Control (PMCC)**

This field controls whether or not PMCs 5 - 6 are included in the Performance Monitor, and the accessibility of groups A and B (see Section 10.4.1) of non-privileged SPRs in problem state as described below.

00 PMCs 5 - 6 are included in the Performance Monitor. Group A is read-only, and group B read access behavior is conditional on MMCR0PMCEXT in problem state. If an attempt is made to write to an SPR in group A in problem state, a Hypervisor Emulation Assistance interrupt will occur.

01 PMCs 5 - 6 are included in the Performance Monitor. Group A is not allowed to be read or written in problem state, and group B is not allowed to be read in problem state. If an attempt is made, in problem state, to read or write to an SPR in group A, or to read from an SPR in group B, a Facility Unavailable interrupt will occur.

10 PMCs 5 - 6 are included in the Performance Monitor. Group A is allowed to be read and written in problem state, and group B except for MMCR1 (SPR 782) is allowed to be read in problem state. If an attempt is made to read MMCR1 in problem state, a Facility Unavailable interrupt will occur.

11 PMCs 5 - 6 are not included in the Performance Monitor. See Section 10.4.2 for details.

**Programming Note**

When the PCR makes SPRs unavailable in problem state, they are treated as undefined, and they are not included in groups A or B regardless of the value of PMCC. Thus when the PCR indicates a version of the architecture prior to V. 2.07 (i.e., PCRv2.06=1), the PMCC field does not affect SPRs MMCR2 or SIER, which are newly-defined in V. 2.07; these SPRs are treated as undefined registers. Accesses to them in problem state result in Hypervisor Emulation Assistance interrupts regardless of the value of PMCC, and Facility Unavailable interrupts do not occur for them. See Section 2.5 for additional information.
been made unavailable by the HFSCR (see Section 7.2.13).

**Programming Note**

In order to give problem state programs the same level of access to the Performance Monitor registers as was specified in Power ISA V 2.06, PMCC must be set to 0b00 (restricting access to read-only) and the PCR should indicate Version 2.06 (restricting access to the set of Performance Monitor SPRs and SPR bits that were defined in V 2.06).

When PMCC=0b00 and a write operation to a Performance Monitor register in group A or B is attempted in problem state, a Hypervisor Emulation Assistance interrupt occurs in order to maintain compatibility with V 2.06. For other values of PMCC, write or read operations to group A and read operations from group B that are not allowed result in Facility Unavailable interrupts. Facility Unavailable interrupts provide the operating system with more information about the type of disallowed access that was attempted than the Hypervisor Emulation Assistance interrupt provides. See Section 7.2.12 for additional information.

**Programming Note**

In order to prevent applications from accessing Performance Monitor registers, PMCC is set to 0b01.

In order to allow applications limited control over the Performance Monitor, PMCC is set to 0b10 or 0b11. These values are also used when Performance Monitor event-based branches are enabled.

### 46:47 Reserved

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 48

**PMC1 Condition Enable** (PMC1CE)

This bit controls whether counter negative conditions due to a negative value in PMC1 are enabled.

0 Counter negative conditions for PMC1 are disabled.

1 Counter negative conditions for PMC1 are enabled.

### 49

**PMCj Condition Enable** (PMCjCE)

This bit controls whether counter negative conditions due to a negative value in any PMCj (i.e., in any PMC except PMC1) are enabled.

0 Counter negative conditions for all PMCjs are disabled.

1 Counter negative conditions for all PMCjs are enabled.

**Trigger** (TRIGGER)

0 The PMCs are incremented (if permitted by other MMCR bits).

1 PMC1 is incremented (if permitted by other MMCR bits). The PMCjs are not incremented until PMC1 is negative or an enabled condition or event occurs, at which time:

- the PMCjs resume incrementing (if permitted by other MMCR bits)
- MMCR0TRIGGER is set to 0

See the description of the FCECE bit, above, regarding the interaction between TRIGGER and FCECE.

**Programming Note**

Uses of TRIGGER include the following.

- Resume counting in the PMCjs when PMC1 becomes negative, without causing a Performance Monitor interrupt. Then freeze all PMCs (and optionally cause a Performance Monitor interrupt) when a PMCj becomes negative. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time a PMCj becomes negative. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=0
  - PMCjCE=1
  - TBEE=0
  - FCECE=1
  - PMAE=1 (if a Performance Monitor interrupt is desired)

- Resume counting in the PMCjs when PMC1 becomes negative, and cause a Performance Monitor interrupt without freezing any PMCs. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time the interrupt handler reads them. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=1
  - TBEE=0
  - FCECE=0
  - PMAE=1
Freeze Counters and BHRB in Problem State Condition (FCPC)

This bit controls the meaning of bit 34 (FCP). See the definition of bit 34 for details.

Programming Note
In order to enable the FCP bit to freeze counters in problem state regardless of MSR_HV, MMCR0_FCPC must be set to 0.

Freeze Counters, sampling and threshold counting in Ultravisor State (FCU)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented, if MSR_SHV PR=0b110.

PMCC Extended (PMCCEXT)

0 If MMCR0_PMCC = b00, a SPR in group B is allowed to be read in problem state.
1 If MMCR0_PMCC = b00 and an attempt is made to read from an SPR in group B, a Facility Unavailable Interrupt will occur.

Control Counters 5 - 6 with Run Latch (CC5-6RUN)

When MMCR0_PMCC = b11, the setting of this bit has no effect; otherwise it is defined as follows.

0 PMCs 5 and 6 are incremented if CTRL_RUN=1 (if permitted by other MMCR bits).
1 PMCs 5 and 6 are incremented regardless of the value of CTRL_RUN (if permitted by other MMCR bits).

Performance Monitor Alert Occurred (PMAO)

0 A Performance Monitor alert has not occurred since the last time software set this bit to 0.
1 A Performance Monitor alert has occurred since the last time software set this bit to 0.

This bit is set to 1 by the hardware when a Performance Monitor alert occurs. This bit can be set to 0 only by the mtspr instruction.

Programming Note
Software can set this bit to 1 and set PMAE to 0 to simulate the occurrence of a Performance Monitor alert.
Software should set this bit to 0 after handling the Performance Monitor alert.

Freeze Counters 1-4 (FC1-4)

0 PMC1 - PMC4 are incremented (if permitted by other MMCR bits).
1 PMC1 - PMC4 are not incremented.

Freeze Counters 5-6 (FC5-6)

0 PMC5 - PMC6 are incremented (if permitted by other MMCR bits).
1 PMC5 - PMC6 are not incremented.

Freeze Counters 1-4 in Wait State (FC1-4WAIT)

0 PMCs 1-4 are incremented (if permitted by other MMCR bits).
1 PMCs 1-4, except for PMCs counting events that are not controlled by this bit, are not incremented if CTRL_RUN=0.

Programming Note
When PMC 1 is counting cycles, it is not controlled by this bit. See the description of the F0 event in Section 10.4.5.

Freeze Counters in Hypervisor State (FCH)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if MSR_SHV PR=0b010.

10.4.5 Monitor Mode Control Register 1

Monitor Mode Control Register 1 (MMCR1) is a 64-bit register as shown below.

Figure 81. Monitor Mode Control Register 1

MMCR1

0

63

MMCR1 enables software to specify the events that are counted by the PMCs.

In the following descriptions, events due to randomly sampled instructions occur only if random sampling is enabled (MMCRASE=1); all other events occur whenever the event specification is met regardless of the value of MMCRASE.

Various events defined below refer to “threshold A” through “threshold H”. The table below specifies the number of threshold event counter events corresponding to each of these thresholds.
The bit definitions of MMCR1 are as follows. Implementation-dependent MMCR1 bits that are not supported are treated as reserved.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Problem state access (SPR 782)</td>
</tr>
<tr>
<td></td>
<td>Privileged access (SPR 782 or 798)</td>
</tr>
<tr>
<td></td>
<td>Implementation-dependent</td>
</tr>
<tr>
<td>32:39</td>
<td><strong>PMC1 Selector</strong> (PMC1SEL)</td>
</tr>
<tr>
<td></td>
<td>The value of PMC1SEL specifies the event to be counted by PMC1 as defined below. All values in the range of E0 - FF that are not specified below are reserved.</td>
</tr>
<tr>
<td>Hex</td>
<td>Event</td>
</tr>
<tr>
<td>00</td>
<td>Disable events. (No events occur.)</td>
</tr>
<tr>
<td>01-BF</td>
<td>Implementation-dependent</td>
</tr>
<tr>
<td>C0-DF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The following events can occur only when random sampling is enabled (MMCRA_{SE}=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in MMCRA_{SM}.)

- **E0** The thread has dispatched a randomly sampled instruction. (RIS)
- **E2** The thread has completed a randomly sampled Branch instruction for which the branch was taken. (RIS, RBS)
- **E4** The thread has failed to locate a randomly sampled instruction in the primary instruction cache. (RIS)
- **E6** The threshold event counter has exceeded the number of events corresponding to threshold A (see Table 6). (RIS, RLS, RBS)
- **E8** The threshold event counter has exceeded the number of events corresponding to threshold E (see Table 6). (RIS, RLS, RBS)

**Table 6: Event Counts for Thresholds A-H**

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4096</td>
</tr>
<tr>
<td>B</td>
<td>32</td>
</tr>
<tr>
<td>C</td>
<td>64</td>
</tr>
<tr>
<td>D</td>
<td>128</td>
</tr>
<tr>
<td>E</td>
<td>256</td>
</tr>
<tr>
<td>F</td>
<td>512</td>
</tr>
<tr>
<td>G</td>
<td>1024</td>
</tr>
<tr>
<td>H</td>
<td>2048</td>
</tr>
</tbody>
</table>

**PMC2 Selector** (PMC2SEL)

The value of PMC2SEL specifies the event to be counted by PMC2 as defined below. All values in the range of E0 - FF that are not specified below are reserved.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Disable events. (No events occur.)</td>
</tr>
<tr>
<td>01-BF</td>
<td>Implementation-dependent</td>
</tr>
<tr>
<td>C0-DF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The following events can occur only when random sampling is enabled (MMCRA_{SE}=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in MMCRA_{SM}.)

- **E0** The thread has obtained the data for a randomly sampled Load instruction from storage that did not reside in any cache. (RIS, RLS)
- **E2** The thread has failed to locate the data for a randomly sampled Load instruction in the primary data cache. (RIS, RLS)
E4 The thread filled a block in the primary data cache with data that were accessed by a randomly sampled Load instruction and obtained from a location other than the secondary or tertiary cache. (RIS, RLS)

E6 The threshold event counter has exceeded the number of events corresponding to threshold B (see Table 6). (RIS, RLS, RBS)

E6 The threshold event counter has exceeded the number of events corresponding to threshold F (see Table 6). (RIS, RLS, RBS)

The following events can occur regardless of whether random sampling is enabled.

F0 The thread has completed a Store instruction to the point at which it has reported all the exceptions it will cause.

F2 The thread has dispatched an instruction.

F4 A cycle has occurred during which the RUN bit of the thread’s CTRL register contained 1.

F6 The thread has failed to locate an ERAT entry during data address translation, and a new ERAT entry corresponding to the data effective address has been written.

F8 An external interrupt for the thread has occurred.

FA The thread has loaded an instruction from a higher level cache than the tertiary cache.

FE The thread has filled a block in the primary data cache with data that were accessed by a Load instruction and obtained from a location other than the secondary or tertiary cache.

**PMC3Selector (PMC3SEL)**

The value of PMC3SEL specifies the event to be counted by PMC3 as defined below. All values in the range of E0 - FF that are not specified below are reserved.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Disable events. (No events occur.)</td>
</tr>
<tr>
<td>01-BF</td>
<td>Implementation-dependent</td>
</tr>
<tr>
<td>C0-DF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The following events can occur only when random sampling is enabled (MMCRA_SE=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in MMCRA_SM.)

E2 The thread has completed a randomly sampled Store instruction to the point at which it has reported all exceptions it will cause. (RIS,RLS)

**PMC4 Selector (PMC4SEL)**

The value of PMC4SEL specifies the event to be counted by PMC4 as defined below. All values in the range of E0 - FF that are not specified below are reserved.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Disable events. (No events occur.)</td>
</tr>
<tr>
<td>01-BF</td>
<td>Implementation-dependent</td>
</tr>
<tr>
<td>C0-DF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
are listed in parentheses. (The sampling mode is specified in MMMRASM.)

E0 The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)
E4 The thread was unable to translate a data virtual address using the TLB for a randomly sampled instruction. (RIS, RLS)
E6 The thread has loaded a randomly sampled instruction from a higher level cache than the tertiary cache. (RIS)
E8 The thread has filled a block in the primary data cache with data that were accessed by a randomly sampled Load instruction and obtained from a location other than the secondary cache. (RIS, RLS)
EA The threshold event counter has exceeded the number of events corresponding to threshold D (see Table 6). (RIS, RLS, RBS)
EC The threshold event counter has exceeded the number of events corresponding to threshold H (see Table 6). (RIS, RLS, RBS)

The following events can occur regardless of whether random sampling is enabled.

F0 The thread has attempted to load data from the primary data cache but no block corresponding to the real address existed.
F2 A cycle has occurred during which the thread has dispatched one or more instructions.
F4 A cycle has occurred during which the PURR was incremented when the RUN bit of the thread’s CTRL register contained 1.
F6 The thread has mispredicted either whether or not the branch would be taken, or if taken, the target address of a Branch instruction.
F8 The thread has discarded prefetched instructions.
FA The thread has completed an instruction when the RUN bit of the thread’s CTRL register contained 1.
FC The thread was unable to translate an instruction virtual address using the TLB, and a new TLB entry corresponding to the instruction virtual address has been written.
FE The thread has obtained the data for a Load instruction from storage that did not reside in any cache.

--- Compatibility Note ---

In versions of the architecture that precede Version 2.02 the PMC Selector Fields were six bits long, and were split between MMMR0 and MMMR1. PMC1-8 were all programmable.

If more programmable PMCs are implemented in the future, additional MMMRs may be defined to cover the additional selectors.

10.4.6 Monitor Mode Control Register 2

Monitor Mode Control Register 2 (MMMR2) is a 64-bit register that contains 9-bit control fields for controlling the operation of PMC1 - PMC6 as shown below.

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>Res’d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>9</td>
<td>17</td>
<td>18</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>35</td>
<td>36</td>
<td>44</td>
<td>45</td>
<td>53</td>
<td>54</td>
<td>63</td>
</tr>
</tbody>
</table>

Figure 82. Monitor Mode Control Register 2

When MMMR0PMCC = 0b11, fields C1 - C4 control the operation of PMC1 - PMC4, respectively and fields C5 and C6 are ignored by the hardware; otherwise, fields C1 - C6 control the operation of PMC1 - PMC6, respectively. The bit definitions of each Cn field are as follows, where n = 1, ..., 6.

When MMMR0PMCC is set to 0b10 or 0b11, providing problem state programs read/write access to MMMR2, only the FCnP0 bits can be accessed. All other bits are not changed when mtspr is executed in problem state, and all other bits return 0s when mfspr is executed in problem state.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Freeze Counter n in Privileged State (FCnP)</td>
</tr>
<tr>
<td>0</td>
<td>PMCn is incremented (if permitted by other MMMR bits).</td>
</tr>
<tr>
<td>1</td>
<td>PMCn is not incremented if MSRHV_PR=0b00.</td>
</tr>
<tr>
<td>1</td>
<td>Freeze Counter n in Problem State if MSRHV=0 (FCnP)</td>
</tr>
<tr>
<td>0</td>
<td>PMCn is incremented (if permitted by other MMMR bits).</td>
</tr>
<tr>
<td>1</td>
<td>PMCn is not incremented if MSRHV_PR=0b01.</td>
</tr>
</tbody>
</table>
Freeze Counter n in Problem State if MSRHV=1 (FCnP1)

0 PMCn is incremented (if permitted by other MMCR bits).
1 PMCn is not incremented if MSRHV PR=0b11.

Freeze Counter n while Mark = 1 (FCnM1)

0 PMCn is incremented (if permitted by other MMCR bits).
1 PMCn is not incremented if MSRPM=1.

Freeze Counter n while Mark = 0 (FCnM0)

0 PMCn is incremented (if permitted by other MMCR bits).
1 PMCn is not incremented if MSRRV PMM=0.

Freeze Counter n in Wait State (FCnWAIT)

0 PMCn is incremented (if permitted by other MMCR bits).
1 PMCn is not incremented if CTRL RUN=0.

Freeze Counter n in Hypervisor State (FCnH)

0 PMCn is incremented (if permitted by other MMCR bits).
1 PMCn is not incremented if MSRRV PR=0b10.

Bits 54:63 of MMCR2 are reserved.

10.4.7 Monitor Mode Control Register A

Monitor Mode Control Register A (MMCRA) is a 64-bit register as shown below.

| MMCRA | 0 | 63 |

The operating system is expected to set CTRL RUN to 0 when the thread is in a "wait state", i.e., when there is no process ready to run.

MMCRA gives privileged programs the ability to control the sampling process, BHRB filtering, and threshold events.

When MMCR0PMCC is set to 0b10 or 0b11, providing problem state programs read/write access to MMCRA, the Threshold Event Counter Exponent (TECX) and Threshold Event Counter Multiplier (TECM) fields are read-only, and all other fields return 0s, when mfspr is executed in problem state; all fields are not changed when mtspr is executed in problem state.

Programming Note

Problem state programs need access to this field in order to enable them to individually enable counters when analyzing sections of code. All the other fields will typically be initialized by the operating system.

Programming Note

Read/write access is provided to MMCRA in problem state (SPR 770) when MMCR0PMCC = 0b10 or 0b11 even though no fields can be modified by mtfspr because future versions of the architecture may allow various fields of MMCRA to be modified in problem state.

The bit definitions of MMCRA are as follows.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:25</td>
<td>Problem state access (SPR 770) Reserved</td>
</tr>
<tr>
<td></td>
<td>Privileged access (SPR 770 or 786) Implementation-dependent</td>
</tr>
<tr>
<td>26</td>
<td>BHRB Recording Disable (BHRBRD)</td>
</tr>
<tr>
<td></td>
<td>This field controls whether BHRB entries are written when BHRB recording is enabled by other bits.</td>
</tr>
<tr>
<td>27:31</td>
<td>Problem state access (SPR 770) Reserved</td>
</tr>
<tr>
<td></td>
<td>Privileged access (SPR 770 or 786) Implementation-dependent</td>
</tr>
<tr>
<td>32:33</td>
<td>BHRB Instruction Filtering Mode (IFM)</td>
</tr>
<tr>
<td></td>
<td>This field controls the filter criterion used by the hardware when recording Branch instructions into the BHRB. See Section 10.5.</td>
</tr>
<tr>
<td>00</td>
<td>All taken Branch instructions are entered into the BHRB unless prevented by other filtering fields.</td>
</tr>
<tr>
<td>01</td>
<td>Only taken calls are entered into the BHRB. Includes direct and indirect: bl, bla, bcl, bcla, bcctl, bcctr, bctar.</td>
</tr>
<tr>
<td>10</td>
<td>Only taken indirect branches (excluding calls) are entered into BHRB: bcrl, bcctr, bctar. This translates to all XL-form taken branches with LK=0.</td>
</tr>
<tr>
<td>11</td>
<td>Only taken conditional branches: bc, bca, bcl, bcla, bclr, bclrl, bcctr, bcctrl, bctar, bctar.</td>
</tr>
</tbody>
</table>
bctarl. This translates to all B-form or XL-form taken branches except those with the BO field encoded to branch always (BO=0b1z1zz).

**Programming Note**

Filtering mode 10 provides additional filtering for unconditional Branch instructions, and for indirect Branch instructions only the target address is recorded.

Filtering mode 11 provides additional filtering for instructions that provide a hint or for which the outcome does not depend on the value of the Condition Register.

### 34:36

**Threshold Event Counter Exponent (TECX)**

This field species the exponent of the threshold event counter value. See Section 10.4.3 for additional information. The maximum exponent supported is at least 5.

### 37

Reserved

### 38:44

**Threshold Event Counter Multiplier (TECM)**

This field species the multiplier of the threshold event counter value. See Section 10.4.3 for additional information.

**Programming Note**

When MMCRO\textsubscript{PMCC} = 0b10 or 0b11, providing problem-state programs read-write access to MMCRA, problem state programs are able to read only the TECX and TECM fields (and are not able to write any fields). The values of these fields are needed during the processing of an event-based branch that occurs due to a counter negative condition for a PMC that was counting “threshold counter exceeded n” events (e.g. MMCRI\textsubscript{PMC1SEL} = 0xE8). Reading these fields enables the application to determine the amount by which the threshold was exceeded. Applications are not given access to other fields, and these other fields must initialized by the operating system.

### 45:47

**Threshold Event Counter Event (TECE)**

This field specifies the event, if any, that is counted by the threshold event counter. The values and meanings are follows.

<table>
<thead>
<tr>
<th>Value</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Disable counting.</td>
</tr>
<tr>
<td>001</td>
<td>A cycle has occurred.</td>
</tr>
<tr>
<td>010</td>
<td>An instruction has completed.</td>
</tr>
<tr>
<td>011</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

All other values are implementation-dependent.

**Threshold Start Event (TS)**

This field specifies the event that causes the threshold event counter to start counting occurrences of the event specified in the Threshold Event Counter Event (TECE) field. The events only occur if MMCRA\textsubscript{SE}=1 (random sampling enabled) and one of the sampling modes listed in parenthesis is in effect. (The sampling mode that is currently in effect is specified in MMCRA\textsubscript{SM}.)

0000 Reserved

0001 The thread has randomly sampled an instruction while it is being decoded. (RIS)

0010 The thread has dispatched a randomly sampled instruction. (RIS)

0011 A randomly sampled instruction has been sent to a facility (e.g. Branch, Fixed Point, etc.) (RIS, RLS, RBS)

0100 The thread has completed a randomly sampled instruction to the point at which it has reported all exceptions it will cause. (RIS, RLS, RBS)

0101 The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)

0110 The thread has failed to locate data for a randomly sampled Load instruction in the primary data cache. (RIS, RLS)

0111 The thread has filled a block in the primary data cache with data that were accessed by a randomly sampled Load instruction. (RIS, RLS)

The definition of the following values depends on whether the access to MMCRA is in problem state or in privileged state.

Problem state access (SPR 770)

1000 - 1111 - Reserved

Privileged access (SPR 770 or 786)

1000 - 1111 - Implementation-dependent

**Threshold End Event (TE)**

This field specifies the event that causes the threshold event counter to stop counting occurrences of the event specified in the Threshold Event Counter Event (TECE) field. The events only occur if MMCRA\textsubscript{SE}=1 (random sampling enabled) and one of the sampling modes listed in parenthesis is in effect. (The sampling mode that is currently in effect is specified in MMCRA\textsubscript{SM}.)

0000 Reserved

0001 The thread has randomly sampled an instruction while it is being decoded. (RIS)
0010 The thread has dispatched a randomly sampled instruction. (RIS)
0011 A randomly sampled instruction has been sent to a facility (e.g. Branch, Fixed Point, etc.) (RIS, RLS, RBS)
0100 The thread has completed a randomly sampled instruction to the point at which it has reported all exceptions that it will cause. (RIS, RLS, RBS)
0101 The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)
0110 The thread has failed to locate data for a randomly sampled Load instruction in the primary data cache. (RIS, RLS)
0111 The thread has filled a block in the primary data cache with data that were accessed by a randomly sampled Load instruction. (RIS, RLS)

The definition of the following values depends on whether the access to MMCRA is in problem state or in privileged state.

Problem state access (SPR 770)
1000 - 1111 - Reserved
Privileged access (SPR 770 or 786)
1000 - 1111 - Implementation-dependent

56 Reserved

57:59 Eligibility for Random Sampling (ES)
When random sampling is enabled (MMCRA$_{SE}$=1) and the SM field indicates random instruction sampling (RIS), the encodings of this field specify the instructions that are eligible to be sampled as follows.
000 All instructions
001 All Load and Store instructions
010 All probe no-op instructions
011 Reserved

The definition of the following values depends on whether the access to MMCRA is in problem state or in privileged state.

Problem state access (SPR 770)
100 - 111 - Reserved
Privileged access (SPR 770 or 786)
100 - 111 - Implementation-dependent

60 Reserved

61:62 Random Sampling Mode (SM)
00 Random Instruction Sampling (RIS) - Instructions that meet the criterion specified in the ES field for random instruction sampling are eligible to be sampled.
01 Random Load/Store Facility Sampling (RLS) - Instructions that meet the criterion specified in the ES field for random Load/Store Facility sampling are eligible for sampling.
10 Random Branch Facility Sampling (RBS) - Instructions that meet the criterion specified in the ES field for random Branch Facility sampling are eligible for sampling.
11 Reserved
**Random Sampling Enable (SE)**

0  Random sampling is disabled.
1  Random sampling is enabled.

See Section 10.4.2.1 for information about random sampling.

### 10.4.8 Sampled Instruction Address Register

The Sampled Instruction Address Register (SIAR) is a 64-bit register.

<table>
<thead>
<tr>
<th>SIAR</th>
<th>//</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>62</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 84. Sampled Instruction Address Register**

When a Performance Monitor alert occurs because of an event caused by execution of a randomly sampled instruction, bits 0:61 of the SIAR contain bits 0:61 of the effective address of the instruction if $\text{SIERSIARV} = 1$ and contains an undefined value if $\text{SIERSIARV} = 0$.

When a Performance Monitor alert occurs because of an event other than an event caused by execution of a randomly sampled instruction, the SIAR contains the effective address of an instruction that was being executed, possibly out-of-order, at or around the time that the Performance Monitor alert occurred.

The instruction located at the effective address contained in the SIAR is called the “sampled instruction”.

Except as described in the next paragraph, the contents of SIAR may be altered by the hardware if and only if $\text{MMCR0PMAE=1}$. Thus after the Performance Monitor alert occurs, the contents of SIAR are not altered by the hardware until software sets $\text{MMCR0PMAE}$ to 1. After software sets $\text{MMCR0PMAE}$ to 1, the contents of SIAR are undefined until the next Performance Monitor alert occurs.

If single step or branch tracing is active ($\text{MSRTE = 0b10}$ or $\text{0b01}$), the contents of SIAR as used by the Performance Monitor facility are undefined and may change even when $\text{MMCR0PMAE=0}$.

**Programming Note**

When the Performance Monitor alert occurs, $\text{SIER_{AMPPR \, SAMPHV}}$ indicates the value of $\text{MSRHV PR}$ that was in effect when the sampled instruction was being executed. (The contents of these SIER bits are visible only in privileged state.)

### 10.4.9 Sampled Data Address Register

The Sampled Data Address Register (SDAR) is a 64-bit register.

<table>
<thead>
<tr>
<th>SDAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

**Figure 85. Sampled Data Address Register**

When a Performance Monitor alert occurs because of an event caused by execution of a randomly sampled instruction, the SDAR contains the effective address of the storage operand of the instruction if $\text{SIERSDARV} = 1$ and contains an undefined value if $\text{SIERSDARV} = 0$.

When a Performance Monitor alert occurs because of an event other than an event caused by execution of a randomly sampled instruction, the SDAR contains the effective address of the storage operand of an instruction that was being executed, possibly out-of-order, at or around the time that the Performance Monitor alert occurred. This storage operand may or may not be the storage operand (if any) of the sampled instruction.

The data located at the effective address contained in the SDAR are called the “sampled data.”

Except as described in the next paragraph, the contents of SDAR may be altered by the hardware if and only if $\text{MMCR0PMAE=1}$. Thus after the Performance Monitor alert occurs, the contents of SDAR are not altered by the hardware until software sets $\text{MMCR0PMAE}$ to 1. After software sets $\text{MMCR0PMAE}$ to 1, the contents of SDAR are undefined until the next Performance Monitor alert occurs.

If single step or branch tracing is active ($\text{MSRTE = 0b10}$ or $\text{0b01}$), the contents of SDAR as used by the Performance Monitor facility are undefined and may change even when $\text{MMCR0PMAE=0}$.

### 10.4.10 Sampled Instruction Event Register

The Sampled Instruction Event Register (SIER) is a 64-bit register.

<table>
<thead>
<tr>
<th>SIER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

**Figure 86. Sampled Instruction Event Register**

When random sampling is enabled and a Performance Monitor alert occurs because of an event caused by execution of a randomly sampled instruction, the SIER contains information about the sampled instruction. The contents of all fields are valid unless otherwise indicated.
When random sampling is disabled or when a Performance Monitor alert occurs because of an event that was not caused by execution of a randomly sampled instruction, the contents of the SIER are undefined.

The contents of SIER may be altered by the hardware if and only if MMCR0.PMAE=1. Thus after the Performance Monitor alert occurs, the contents of SIER are not altered by the hardware until software sets MMCR0.PMAE to 1. After software sets MMCR0.PMAE to 1, the contents of SIER are undefined until the next Performance Monitor alert occurs.

The bit definitions of the SIER are as follows.

0:37 The definition of these bits depends on whether the access to SIER is in problem state or in privileged state.

Problem state access (SPR 768)
Reserved
Privileged access (SPR 768 or 784)
Implementation-dependent

38:40 The definition of these bits depends on whether the access to SIER is in problem state or in privileged state.

Problem state access (SPR 768)
Reserved
Privileged access (SPR 768 or 784)

38 Sampled MSRPR (SAMPPR)
Value of MSRPR when the Performance Monitor alert occurred.

39 Sampled MSRHV (SAMPHV)
Value of MSRHV when the Performance Monitor alert occurred.

40 Reserved

41 SIAR Valid (SIARV)
Set to 1 when the contents of the SIAR are valid (i.e., they contain the effective address of the sampled instruction); otherwise set to 0.

42 SDAR Valid (SDARV)
Set to 1 when the contents of the SDAR are valid (i.e., they contain the effective address of the sampled instruction); otherwise set to 0.

43 Threshold Exceeded (TE)
Set to 1 by the hardware if the contents of the threshold event counter exceeded the maximum value when the Performance Monitor alert occurred; otherwise set to 0 by the hardware.

44 Slew Down
Set to 1 by the hardware if the processor clock was lower than nominal when the Performance Monitor alert occurred; otherwise set to 0 by the hardware.

45 Slew Up
Set to 1 by the hardware if the processor clock was higher than nominal when the Performance Monitor alert occurred; otherwise set to 0 by the hardware.

46:48 Sampled Instruction Type (SITYPE)
This field indicates the sampled instruction type. The values and their meanings are as follows.

000 The hardware is unable to indicate the sampled instruction type
001 Load Instruction
010 Store instruction
011 Branch Instruction
100 Floating-Point Instruction other than a Load or Store instruction
101 Fixed-Point Instruction other than a Load or Store instruction
110 Condition Register or System Call instruction
111 Reserved

49:51 Sampled Instruction Cache Information (SICACHE)
This field provides cache-related information about the sampled instruction.

000 The hardware is unable to provide any cache-related information for the sampled instruction.
001 The thread obtained the instruction in the primary instruction cache.
010 The thread obtained the instruction in the secondary cache.
011 The thread obtained the instruction in the tertiary cache.
100 The thread failed to obtain the instruction in the primary, secondary, or tertiary cache
101 Reserved
110 Reserved
111 Reserved

52 Sampled Instruction Taken Branch (SITAKBR)
Set to 1 if the SITYPE field indicates a Branch instruction and the branch was taken; otherwise set to 0.

53 Sampled Instruction Mispredicted Branch (SIMISPRED)
Set to 1 if the SITYPE field indicates a Branch instruction and the thread has mispredicted either whether or not the branch would be taken, or if taken, the target address; otherwise set to 0.

54:55  **Sampled Branch Instruction Misprediction Information** (SIMISPREDI)

If SIMISPRED=1, this field indicates how the thread mispredicted the outcome of a Branch instruction; otherwise this field is set to 0s.

00  The instruction was not a mispredicted Branch instruction.
01  The thread mispredicted whether or not the branch would be taken because the contents of the Condition Register differed from the predicted contents.
10  The thread mispredicted the target address of the instruction.
11  Reserved

56  **Sampled Instruction Data ERAT Miss (SIDERAT)**

When the SITYPE field indicates a Load or Store instruction, this field is set to 1 if the thread has failed to locate an ERAT entry during data address translation for the sampled instruction and otherwise is set to 0.

When the SITYPE field does not indicate a Load or Store instruction, the contents of this field are undefined.

57:59  **Sampled Instruction Data Address Translation Information** (SIDAXLATE)

This field contains information about data address translation for the sampled instruction. If multiple data address translations were performed, the information pertains to the last translation. The values and their meanings are as follows.

000  The instruction did not require data address translation.
001  The thread translated the data virtual address using the TLB.
010  A PTEG required for data address translation for the instruction was obtained from the secondary cache.
011  A PTEG required for data address translation for the instruction was obtained from the tertiary cache.
100  A PTEG required for data address translation for the instruction was obtained from storage that did not reside in any cache.
101  A PTEG required for data address translation for the instruction was obtained from a cache on a different multi-threaded processor that resides on the same chip as the thread.
110  A PTEG required for data address translation for the instruction was obtained from a cache on a different chip from the thread.
111  Reserved

60:62  **Sampled Instruction Data Storage Access Information** (SIDSATI)

This field contains information about data storage accesses made by the sampled instruction. The values and their meanings are as follows.

000  The instruction did not require data address translation.
001  The instruction was a Read for which the thread obtained the referenced data from the primary data cache.
010  The instruction was a Read for which the thread obtained the referenced data from the secondary cache.
011  The instruction was a Read for which the thread obtained the referenced data from the tertiary cache.
100  The instruction was a Read for which the thread obtained the referenced data from storage that did not reside in any cache.
101  The instruction was a Read for which the thread obtained the referenced data from a cache on a different multi-threaded processor that resides on the same chip as the thread.
110  The instruction was a Read for which the thread obtained the referenced data from a cache on a different chip from the thread.
111  The instruction was a Store for which the data were placed into a location other than the primary data cache.

63  **Sampled Instruction Completed** (SICMPL)

Set to 1 if the sampled instruction has completed; otherwise set to 0.

10.4.11 Other Performance Monitor Registers

SIER2, SIER3 and MMCR3 are 64-bit registers. The contents of these registers are implementation-dependent.
The Branch History Rolling Buffer (BHRB) is described in Chapter 7 of Book II but only at the level required by application programmers. Additional aspects of the BHRB are described here.

In order to enable problem state programs to use the BHRB, MMCR0BHRBA must be set to 1 to enable execution of clrbhrb and mfbhrbe instructions in problem state. Additionally, MMCR0PMCC must be set to 0b10 or 0b11 to allow problem state programs to read and write the necessary Performance Monitor registers. (See Section 10.4.4.)

If Performance Monitor event-based branching is desired, MMCR0EBE must also be set to 1 to enable Performance Monitor event-based branches.

### Programming Note

Applications are expected to access the contents of SIER2, SIER3 and MMCR3 by means of a privileged service program which will be typically invoked by a system call and which is capable of interpreting the contents of these registers, and returns to the application a generalized summary of the register contents in a form that is not implementation-dependent.

The recommended secure configuration of the Performance Monitor Facility is to disable access to group B Performance Monitor registers in problem state via setting MMCR0PMCC and MMCR0PMCC\_CEXT appropriately, which is congruous to the programming model described above.

### 10.5 Branch History Rolling Buffer

The Branch History Rolling Buffer (BHRB) is described in Chapter 7 of Book II but only at the level required by application programmers. Additional aspects of the BHRB are described here.

In order to enable problem state programs to use the BHRB, MMCR0BHRBA must be set to 1 to enable execution of clrbhrb and mfbhrbe instructions in problem state. Additionally, MMCR0PMCC must be set to 0b10 or 0b11 to allow problem state programs to read and write the necessary Performance Monitor registers. (See Section 10.4.4.)

If Performance Monitor event-based branching is desired, MMCR0EBE must also be set to 1 to enable Performance Monitor event-based branches.

### 10.5.1 BHRB Filtering

When the BHRB is written by hardware, only those Branch instructions that meet the filtering criterion specified in MMCRA\_IFM and for which the branch was taken are included.

Filtering restricts the type of Branch instructions that are entered into the BHRB. The filtering criteria are defined using the following terminology.

- **Call**: A Branch instruction with the LK field set to 1.
- **Return**: A \texttt{bclr} instruction with the BH field set to 0s.
- **Jump**: Any Branch instruction that is not a call or a return.
- **Conditional Branch**: Any Branch instruction other than an I-Form Branch instruction, or a B- or XL-Form Branch instruction with the BO field set to “branch always.” (See Figure 40 in Book I.)
- **Unconditional Branch**: Any Branch instruction other than a conditional branch instruction
- **Indirect Branch**: Any XL-Form Branch instruction
- **Direct Branch**: Any B- or I-Form Branch instruction

Software is able to prevent various combinations of each of the above types of Branch instructions from being entered into the BHRB using the IFM field in MMCRA. (See Section 10.4.7, “Monitor Mode Control Register A”.)

The BHRB is written by the hardware only in problem state (MSR\_PR=1), and if and only if MMCR0PMAC=1 and MMCRABHRBRD=0. After MMCR0PMAC has been set to 1 and a Performance Monitor alert occurs, MMCR0PMAC is set to 0 and the BHRB is not altered by hardware until software sets MMCR0PMAC to 1 again.

When MMCR0PMAC=1, \texttt{mfbhbrbe} instructions return 0s to the target register.

### Programming Note

\texttt{mfbhbrbe} instructions return 0s when MMCR0PMAE=1 in order to prevent software from reading the BHRB while it is being written by hardware.
Chapter 11. Processor Control

11.1 Overview

The Processor Control facility provides a mechanism for the ultravisor or hypervisor to send messages to other threads in the system. Privileged non-hypervisor programs are able to send messages to other threads on the same multi-threaded processor; however if the processor is configured into sub-processors, privileged non-hypervisor programs can only send messages to other threads on the same sub-processor.

11.2 Programming Model

Ultravisor-level, hypervisor-level, and privileged-level messages can be sent. Ultravisor-level messages are sent using the `msgsndu` instruction and cause ultravisor-level exceptions when received. Hypervisor-level messages are sent using the `msgsnd` instruction and cause hypervisor-level exceptions when received. Privileged-level messages are sent using the `msgsndp` instruction and cause privileged-level exceptions when received. For all three instructions, the message type and destination threads are specified in a General Purpose Register.

If a message is received by a thread, the exception corresponding to the message type is generated. When the exception is generated, the corresponding interrupt occurs when no higher priority exception exists and the interrupt is enabled (MSR\textsubscript{EE}=1 for the Directed Privileged Doorbell interrupt, MSR\textsubscript{EE}=1 or MSR\textsubscript{HV}=0 for the Directed Hypervisor Doorbell interrupt, and MSR\textsubscript{EE}=1 or MSR\textsubscript{3 HV PR}=0b110 for the Directed Ultravisor Doorbell interrupt).

A Directed Privileged Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a `mtspr(DPDES)` or `msgclrp` instruction.

A Directed Hypervisor Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a `msgclrp` instruction.

A Directed Ultravisor Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a `msgclru` instruction.

If a Doorbell exception of a given privilege is present and the corresponding interrupt is pended because MSR\textsubscript{EE}=0, additional Doorbell exceptions of that privilege are ignored until the exception is cleared.

11.3 Processor Control Registers

11.3.1 Directed Privileged Doorbell Exception State

The layout of the Directed Privileged Doorbell Exception State (DPDES) register is shown in Figure 87.

![Figure 87. Directed Privileged Doorbell Exception State Register](image)

The DPDES register is a 64-bit register. For \( t < T \), where \( T \) is the number of threads on the sub-processor (or on the multi-threaded processor if sub-processors are not supported), bit 63-\( t \) corresponds to the thread with privileged thread number \( t \).

The value of bit \( t \) indicates the presence of a Directed Privileged Doorbell exception on the thread with privileged thread number \( t \). Bit \( t \) is cleared when a Directed Privileged Doorbell interrupt occurs on thread \( t \).

When the contents of DPDES\( _{63-t} \) change from 0 to 1, a Directed Privileged Doorbell exception will come into existence on privileged thread number \( t \) within a reasonable period of time. When the contents of DPDES\( _{63-t} \) change from 1 to 0, the existing Directed Privileged Doorbell exception, if any, on privileged thread number \( t \), will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event on privileged thread number \( t \).

The preceding paragraph applies regardless of whether the change in the contents of DPDES\( _{63-t} \) is the result a `msgsndp` or `msgclrp` instruction or of modification of the DPDES register caused by execution of an `mtspr` (DPDES) instruction.

Bits 0:63-\( T \) of the DPDES are reserved.
The primary use of the DPDES is to provide the means for the hypervisor to save a [sub-]processor's Directed Privileged Doorbell exception state when the set of programs running on the [sub-]processor is swapped out or moved from one [sub-]processor to another. Since there is no such need for a similar function for the hypervisor or ultravisor, there is no similar register for the hypervisor or ultravisor. Privileged programs are able to read the DPDES in order to poll for Directed Privileged Doorbell exceptions when the corresponding interrupt is disabled (MSR$_{EE}$=1).
11.4 Processor Control Instructions

`msgsndu`, `msgsnd`, `msgsndp`, `msgclru`, `msgclr`, and `msgclrp` instructions are provided for sending and clearing messages. `msgsnc` is provided to enable the thread that is target of a `msgsndu` or `msgsnd` instruction to ensure that stores performed by the message-sending thread before it executed `msgsndu` or `msgsnc` have been performed with respect to the target thread. `msgsndp` and `msgclrp` are privileged instructions; `msgsnd`, `msgclru`, and `msgsnc` are hypervisor privileged instructions; `msgsndu` and `msgclrp` are ultravisor privileged instructions.

**Message Send Ultravisor X-form**

```plaintext
msgsndu  RB

<table>
<thead>
<tr>
<th>31</th>
<th>///</th>
<th>11</th>
<th>RB</th>
<th>78</th>
<th>/</th>
</tr>
</thead>
</table>

msgtype ← GPR(RB)32:36
payload ← GPR(RB)37:63
if (msgtype = 0x05) then
  send_msg(msgtype, payload)
```

`msgsndu` sends a message to other threads in the system. The message type and destination thread(s) are specified in RB.

**RB**

```plaintext
0 6 11 16 21 31
```

`<Message Payload->`

<table>
<thead>
<tr>
<th>///</th>
<th>TYPE</th>
<th>B</th>
<th>///</th>
<th>PROCIDTAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>37</td>
<td>39</td>
<td>44</td>
</tr>
</tbody>
</table>

**Figure 88. RB Contents for `msgsndu`**

The contents of RB are defined below. Bits 37:63 are referred to as the message payload.

**Field**

**Description**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Reserved</td>
</tr>
<tr>
<td>32:36</td>
<td>Type</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If Type=0x05, then a Directed Ultravisor Doorbell message is to be sent to the thread(s) specified in the Message Payload field. All other values of the Type field are reserved; if the instruction is executed with this field set to a reserved value, the instruction is treated as a no-op.</td>
</tr>
<tr>
<td>37:38</td>
<td>Broadcast (B)</td>
</tr>
<tr>
<td>00</td>
<td>The message is sent to the thread for which PIR44:63 is equal to the value of the PROCIDTAG field in the message payload.</td>
</tr>
<tr>
<td>01</td>
<td>The message is sent to all threads on the same sub-processor as the thread for which PIR44:63 is equal to the value of the PROCIDTAG field in the message payload.</td>
</tr>
</tbody>
</table>

10 The message is sent to all threads on the same multi-threaded processor as the thread for which PIR44:63 is equal to the value of the PROCIDTAG field in the message payload.

11 Reserved

39:43 Reserved

44:63 PROCIDTAG

This field indicates the recipient thread(s) as specified in the B field. If this field set to a value that is not the same as bits PIR44:63 of any thread in the system, then the instruction behaves as if it were a no-op.

The actions taken on receipt of a message are defined in Section 11.2.

This instruction is ultravisor privileged.

**Special Registers Altered:**

None

**Programming Note**

If `msgsndu` is used to notify the receiver that updates have been made to storage, a `sync` should be placed between the stores and the `msgsndu`. See Section 6.9.2.
**Message Clear Ultravisor**

\[ \text{msgclru} \hspace{1em} \text{RB} \]

- \( t \) is set to hypervisor thread number of executing thread
- if \((\text{msgtype} = 0x05)\) then
  - clear any Directed Ultravisor Doorbell exception for thread \( t \)

\textit{msgclru} clears a message previously accepted by the thread executing the \textit{msgclru}.

Let \( \text{msgtype} \) be \((\text{RB})_{32:36}\), and let \( t \) be the hypervisor thread number of the thread executing the \textit{msgclru} instruction.

If \( \text{msgtype} = 0x05 \), then clear any Directed Ultravisor Doorbell exception that exists on thread \( t \); otherwise, this instruction is treated as a no-op.

This instruction is ultravisor privileged.

**Special Registers Altered:**

- None

---

**Programming Note**

\textit{msgclru} is typically issued only when MSR\_EE=0. If \textit{msgclru} is executed when MSR\_EE=1 when a Directed Ultravisor Doorbell interrupt is about to occur, the corresponding interrupt may or may not occur.

**Message Send**

\[ \text{msgsnd} \hspace{1em} \text{RB} \]

- \( \text{msgtype} \) is set to \( \text{GPR}(\text{RB})_{32:36} \)
- \( \text{payload} \) is set to \( \text{GPR}(\text{RB})_{37:63} \)
- if \((\text{msgtype} = 0x05)\) then
  - \( \text{send\_msg}(\text{msgtype}, \text{payload}) \)

\textit{msgsnd} sends a message to other threads in the system. The message type and destination thread(s) are specified in \text{RB}.

**RB**

\[ \text{// Message Payload -} \]

- Field Description
  - 0:31 Reserved
  - 32:36 Type
    - If Type=0x05, then a Directed Hypervisor Doorbell message is to be sent to the thread(s) specified in the Message Payload field.
    - All other values of the Type field are reserved; if the instruction is executed with this field set to a reserved value, the instruction is treated as a no-op.
  - 37:38 Broadcast (B)
    - 00 The message is sent to the thread for which PIR\_44:63 is equal to the value of the PROCIDTAG field in the message payload.
    - 01 The message is sent to all threads on the same sub-processor as the thread for which PIR\_44:63 is equal to the value of the PROCIDTAG field in the message payload.
    - 10 The message is sent to all threads on the same multi-threaded processor as the thread for which PIR\_44:63 is equal to the value of the PROCIDTAG field in the message payload.
    - 11 Reserved
  - 39:43 Reserved
  - 44:63 PROCIDTAG
    - This field indicates the recipient thread(s) as specified in the B field. If this field set to a
value that is not the same as bits PIR44:63 of any thread in the system, then the instruction behaves as if it were a no-op.

The actions taken on receipt of a message are defined in Section 11.2.

This instruction is hypervisor privileged.

**Special Registers Altered:**
None

**Programming Note**

If `msgsnd` is used to notify the receiver that updates have been made to storage, a `sync` should be placed between the stores and the `msgsnd`. See Section 6.9.2.

---

### Message Clear

`msgclr` clears a message previously accepted by the thread executing the `msgclr`.

Let `msgtype` be (RB)32:36, and let `t` be the hypervisor thread number of the thread executing the `msgclr` instruction.

If `msgtype` = 0x05, then clear any Directed Hypervisor Doorbell exception that exists on thread `t`; otherwise, this instruction is treated as a no-op.

This instruction is hypervisor privileged.

**Special Registers Altered:**
None

**Programming Note**

`msgclr` is typically issued only when MSR_EE=0. If `msgclr` is executed when MSR_EE=1 when a Directed Hypervisor Doorbell interrupt is about to occur, the corresponding interrupt may or may not occur.
**Message Send Privileged X-form**

```
msgsndp  RB
0 31  ///  /// RB 142 ///
```

```
msgtype ← (RB)32:36
payload ← (RB)37:63
t ← (RB)57:63
if msgtype = 5 and 
t ≤ maximum privileged thread number 
on processor or sub-processor 
then
  DPDES63 ← 1
  send_msg(msgtype, payload, t)
```

`msgsndp` sends a message to other threads that are on the same multi-threaded processor (if the processor is not in sub-processor mode) or to other threads that are on the same sub-processor (if the processor is in sub-processor mode). The message type and destination thread(s) are specified in RB.

**RB**

```
Message Payload
/// TYPE /// TIRTAG
0 32 37 39 57 63
```

**Figure 90. RB Contents for `msgsndp`**

The contents of RB are defined below. Bits 37:63 are referred to as the message payload.

**Bits Description**

- **37:56** Reserved
- **57:63** TIRTAG

This message is sent to the thread for which the privileged thread number is equal to contents of the TIRTAG field of the message payload, and one of the following conditions applies:

- for processors that are not partitioned into sub-processors, the thread is sent to the thread on the same multi-threaded processor for which the privileged thread number is equal to the contents of the TIRTAG field of the message payload.
- for processors that are partitioned into sub-processors, the thread is sent to the thread on the same sub-processor for which the privileged thread number is equal to the contents of the TIRTAG field of the message payload.

If `msgsndp` is executed with TIRTAG set to a value greater than the highest privileged thread number on the sub-processor (or on the multi-threaded processor if sub-processors are not supported), then this instruction behaves as a no-op.

The actions taken on receipt of a message are defined in Section 11.2.

This instruction is privileged.

**Special Registers Altered:**

- DPDES

**Programming Note**

If `msgsndp` is used to notify the receiver that updates have been made to storage, a `lwsync` or `sync` should be placed between the stores and the `msgsndp`. See Section 6.9.2.
Message Clear Privileged   X-form

msgclrp | RB
--- | ---
31 | /// | /// | RB | 174 | ///
0 | 8 | 11 | 16 | 21 | 31

msgclrp clears a message previously accepted by the thread executing the msgclrp.

Let msgtype be (RB)_{32:36}, and let t be the privileged thread number of the thread executing the msgclrp.

If msgtype = 0x05, then clear any Directed Privileged Doorbell exception that exists on thread t by setting DPDES_{63-t} to 0; otherwise, this instruction is treated as a no-op.

This instruction is privileged.

Special Registers Altered:

- DPDES

Programming Note

msgclrp is typically issued only when MSR_{EE}=0. If msgclrp is executed when MSR_{EE}=1 when a Directed Hypervisor Doorbell interrupt is about to occur, the corresponding interrupt may or may not occur.

Message Synchronize   X-form

msgsync

0 | /// | /// | /// | 886 | ///
0 | 8 | 11 | 16 | 21 | 31

In conjunction with the Synchronize and msgsnd or msgsndu instructions, the msgsync instruction provides an ordering function for stores that have been performed with respect to the thread executing the Synchronize and msgsnd or msgsndu instructions, relative to data accesses by other threads that are performed after a Directed Ultravisor Doorbell or Directed Hypervisor Doorbell interrupt has occurred, as described in the Synchronize instruction description on p. 1218.

This instruction is hypervisor privileged.

Special Registers Altered:

- None

Programming Note

When used in conjunction with msgsndu or msgsnd, Synchronize with L = 0 or 2 is executed on the thread that will execute the msgsndu or msgsnd, and msgsync is executed on another thread -- typically the thread that is the target of the msgsndu or msgsnd, but possibly any other thread (partly because the software that services the Directed Ultravisor Doorbell or Directed Hypervisor Doorbell interrupt may ultimately run on a thread other than that which received the exception). The Synchronize precedes the msgsndu or msgsnd, the msgsync is executed after the Directed Ultravisor Doorbell or Directed Hypervisor Doorbell interrupt occurs, and precedes all instructions that need to "see" the values stored by the stores that are in set A of the memory barrier created by the Synchronize; see Section 6.9.2, "Synchronize Instruction".
Chapter 12. Synchronization Requirements for Context Alterations

Changing the contents of certain System Registers, the contents of SLB entries, or the contents of other system resources that control the context in which a program executes can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed and data accesses are performed. For example, changing MSRIR from 0 to 1 has the side effect of enabling translation of instruction addresses. These side effects need not occur in program order, and therefore may require explicit synchronization by software. (Program order is defined in Book II.)

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed or data accesses are performed, is called a context-altering instruction. This chapter covers all the context-altering instructions. The software synchronization required for them is shown in Table 7 (for data access) and Table 8 (for instruction fetch and execution).

The notation “CSI” in the tables means any context synchronizing instruction (e.g., sc, isync, or rfid). A context synchronizing interrupt (i.e., any interrupt except non-recoverable System Reset or non-recoverable Machine Check) can be used instead of a context synchronizing instruction. If it is, phrases like “the synchronizing instruction”, below, should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required before (after) a context-altering instruction, “the synchronizing instruction before (after) the context-altering instruction” should be interpreted as meaning the context-altering instruction itself.

The synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. The synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

### Programming Note

Sometimes advantage can be taken of the fact that certain events, such as interrupts, and certain instructions that occur naturally in the program, such as the rfid that returns from an interrupt handler, provide the required synchronization.

Because the instructions between the first synchronizing instruction (exclusive) and the second synchronizing instruction (inclusive) may be fetched or executed in either context, if the context alteration affects whether the second synchronizing instruction can be fetched or executed then the context alteration will not necessarily be synchronized in the manner the programmer expected. For example, if the second synchronizing instruction is in a different virtual page from the context-altering instruction, and fetching instructions from this virtual page is prohibited by Virtual Page Class Key Storage Protection, and the context-altering instruction is an mtiamr that enables fetching instructions from this virtual page, it is indeterminate whether the second synchronizing instruction will be executed or a [Hypervisor] Instruction Storage interrupt will occur instead.

No software synchronization is required before or after a context-altering instruction that is also context synchronizing or when altering the MSR in most cases (see the tables). No software synchronization is required before most of the other alterations shown in Table 8, because all instructions preceding the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the hardware must determine whether any of these preceding instructions are context synchronizing).

Unless otherwise stated, the material in this chapter assumes a single-threaded environment.
<table>
<thead>
<tr>
<th>Instruction or Event</th>
<th>Required Before</th>
<th>Required After</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfid</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>hrfid</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>urfid</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfscv</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>scv</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>Trap</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtspr (AMR)</td>
<td>CSI</td>
<td>CSI</td>
<td>13</td>
</tr>
<tr>
<td>mtspr (PIDR)</td>
<td>CSI</td>
<td>CSI</td>
<td>6,20</td>
</tr>
<tr>
<td>mtspr (DAWRn)</td>
<td>CSI</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtspr (DAWRXn)</td>
<td>CSI</td>
<td>CSI</td>
<td>11,17</td>
</tr>
<tr>
<td>mtspr (HRMOR)</td>
<td>CSI</td>
<td>CSI</td>
<td>11,17</td>
</tr>
<tr>
<td>mtspr (URMOR)</td>
<td>CSI</td>
<td>CSI</td>
<td>11,17</td>
</tr>
<tr>
<td>mtspr (LPCR)</td>
<td>CSI</td>
<td>CSI</td>
<td>14,19</td>
</tr>
<tr>
<td>mtspr (PTCR)</td>
<td>ptesync</td>
<td>CSI</td>
<td>3</td>
</tr>
<tr>
<td>mtspr (SMFCTRL)</td>
<td>CSI</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtsmsrd (SF)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtsmsr[d] (PR)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtsmsr[d] (DR)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtspr (LPIDR)</td>
<td>CSI</td>
<td>CSI</td>
<td>6,14,20</td>
</tr>
<tr>
<td>slbie</td>
<td>CSI</td>
<td>CSI</td>
<td>4</td>
</tr>
<tr>
<td>slbieg</td>
<td>CSI</td>
<td>CSI</td>
<td>4,6</td>
</tr>
<tr>
<td>slbia</td>
<td>CSI</td>
<td>CSI</td>
<td>4</td>
</tr>
<tr>
<td>slbmtte</td>
<td>CSI</td>
<td>CSI</td>
<td>4,10</td>
</tr>
<tr>
<td>tlibie</td>
<td>CSI</td>
<td>CSI</td>
<td>4,6</td>
</tr>
<tr>
<td>tlibiel</td>
<td>CSI</td>
<td>ptesync</td>
<td>4</td>
</tr>
<tr>
<td>Store(PTE)</td>
<td>none</td>
<td>(ptesync, CSI)</td>
<td>5,6</td>
</tr>
<tr>
<td>Store(STE)</td>
<td>none</td>
<td>(ptesync, CSI)</td>
<td>5,6</td>
</tr>
<tr>
<td>Store(PRTE)</td>
<td>none</td>
<td>(ptesync, CSI)</td>
<td>5,6</td>
</tr>
<tr>
<td>Store(PATE)</td>
<td>none</td>
<td>(ptesync, CSI)</td>
<td>5,6,19</td>
</tr>
</tbody>
</table>

Table 7: Synchronization requirements for data access

<table>
<thead>
<tr>
<th>Instruction or Event</th>
<th>Required Before</th>
<th>Required After</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfid</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>hrfid</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>urfid</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfscv</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>scv</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>Trap</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtsmsrd (SF)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtsmsr[d] (EE)</td>
<td>none</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>mtsmsr[d] (PR)</td>
<td>none</td>
<td>none</td>
<td>8</td>
</tr>
<tr>
<td>mtsmsr[d] (FP)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtsmsr[d] (FE0,FE1)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtsmsr[d] (TE)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtsmsr[d] (IR)</td>
<td>none</td>
<td>none</td>
<td>8</td>
</tr>
<tr>
<td>mtsmsr[d] (RI)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtspr (DEC)</td>
<td>none</td>
<td>none</td>
<td>9</td>
</tr>
<tr>
<td>mtspr (PIDR)</td>
<td>CSI</td>
<td>CSI</td>
<td>6</td>
</tr>
<tr>
<td>mtspr (IAMR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtspr (CTRL)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtspr (FSCR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtspr (DPDES)</td>
<td>none</td>
<td>CSI</td>
<td>17</td>
</tr>
<tr>
<td>mtspr (CIABR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtspr (HFSCR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtspr (HDEC)</td>
<td>none</td>
<td>none</td>
<td>9</td>
</tr>
<tr>
<td>mtspr (HRMOR)</td>
<td>none</td>
<td>CSI</td>
<td>8,11,17</td>
</tr>
<tr>
<td>mtspr (URMOR)</td>
<td>none</td>
<td>CSI</td>
<td>8,11,17</td>
</tr>
<tr>
<td>mtspr (LPCR)</td>
<td>none</td>
<td>CSI</td>
<td>12,14,19</td>
</tr>
<tr>
<td>mtspr (LPIDR)</td>
<td>CSI</td>
<td>CSI</td>
<td>6,14,17</td>
</tr>
<tr>
<td>mtspr (PCR)</td>
<td>none</td>
<td>CSI</td>
<td>17</td>
</tr>
<tr>
<td>mtspr (PTCR)</td>
<td>ptesync</td>
<td>CSI</td>
<td>3,17</td>
</tr>
<tr>
<td>mtspr (SMFCTRL)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtspr (Perf. Mon.)</td>
<td>none</td>
<td>CSI</td>
<td>15,18</td>
</tr>
<tr>
<td>mtspr (BESC)</td>
<td>none</td>
<td>CSI</td>
<td>16,18</td>
</tr>
<tr>
<td>slbie</td>
<td>none</td>
<td>CSI</td>
<td>4</td>
</tr>
<tr>
<td>slbieg</td>
<td>none</td>
<td>CSI</td>
<td>4</td>
</tr>
<tr>
<td>slbia</td>
<td>none</td>
<td>CSI</td>
<td>4,6</td>
</tr>
<tr>
<td>slbmtte</td>
<td>none</td>
<td>CSI</td>
<td>4,8,10</td>
</tr>
<tr>
<td>tlibie</td>
<td>none</td>
<td>CSI</td>
<td>4,6</td>
</tr>
<tr>
<td>tlibiel</td>
<td>none</td>
<td>CSI</td>
<td>4</td>
</tr>
<tr>
<td>Store(PTE)</td>
<td>none</td>
<td>(ptesync, CSI)</td>
<td>5,6,8</td>
</tr>
<tr>
<td>Store(STE)</td>
<td>none</td>
<td>(ptesync, CSI)</td>
<td>5,6,8</td>
</tr>
<tr>
<td>Store(PRTE)</td>
<td>none</td>
<td>(ptesync, CSI)</td>
<td>5,6,8</td>
</tr>
<tr>
<td>Store(PATE)</td>
<td>none</td>
<td>(ptesync, CSI)</td>
<td>5,6,8,19</td>
</tr>
</tbody>
</table>

Table 8: Synchronization requirements for instruction fetch and/or execution
Notes:

1. The effect of changing the EE bit is immediate, even if the mtmsr[d] instruction is not context synchronizing (i.e., even if L=1).
   - If an mtmsr[d] instruction sets the EE bit to 0, neither an External interrupt, a Decrementer interrupt nor a Performance Monitor interrupt occurs after the mtmsr[d] is executed.
   - If an mtmsr[d] instruction changes the EE bit from 0 to 1 when an External, Decrementer, Performance Monitor or higher priority exception exists, the corresponding interrupt occurs immediately after the mtmsr[d] is executed, and before the next instruction is executed in the program that set EE to 1.
   - If a hypervisor executes the mtmsr[d] instruction that sets the EE bit to 0, a Hypervisor Decrementer interrupt does not occur after mtmsr[d] is executed as long as the thread remains in hypervisor state.
   - If the hypervisor executes an mtmsr[d] instruction that changes the EE bit from 0 to 1 when a Hypervisor Decrementer or higher priority exception exists, the corresponding interrupt occurs immediately after the mtmsr[d] instruction is executed, and before the next instruction is executed, provided HDICE is 1.

2. Synchronization requirements for this instruction are implementation-dependent.

3. The PTCR controls all implicit and explicit storage accesses performed by all threads on the processor when the thread is not in hypervisor or ultravisor real addressing mode. Modifying the PTCR requires that the following conditions be achieved on all threads on the processor:
   - the thread is in hypervisor or ultravisor real addressing mode
   - all previous accesses (implicit and explicit) initiated when the thread was not in hypervisor or ultravisor real addressing mode have been performed with respect to all threads
   - no subsequent accesses which require translation have been initiated

4. For data accesses, the context synchronizing instruction before the slbie, slbief, slbia, slbmt, tlbie, or tlbief instruction ensures that all preceding instructions that access data storage have completed to a point at which they have reported all exceptions they will cause.

The context synchronizing instruction after the slbie, slbief, slbia, tlbie or tlbief instruction ensures that storage accesses associated with instructions following the context synchronizing instruction will not use the SLB entry(s), TLB entry(s), or implementation-specific lookaside information being invalidated.

(For tlbief and tlbie, if it is necessary to order storage accesses associated with preceding instructions, or Reference and Change bit updates associated with preceding address translations, with respect to subsequent data accesses, a ptesync instruction must also be used, either before or after the tlbie or tlbief instruction. These effects of the ptesync instruction are described in the last paragraph of Note 5.)

5. The notation “(ptesync,CSI)” denotes an instruction sequence. Other instructions may be interleaved with this sequence, but these instructions must appear in the order shown.

No software synchronization is required before the Store instruction because (a) stores are not performed out-of-order and (b) address translations associated with instructions preceding the Store instruction are not performed again after the store has been performed (see Section 6.5). These properties ensure that all address translations associated with instructions preceding the Store instruction will be performed using the old contents of the PTE.

The ptesync instruction after the Store instruction ensures that all searches of the Page Table that are performed after the ptesync instruction completes will use the value stored (or a value stored subsequently). The context synchronizing instruction after the ptesync instruction ensures that any address translations associated with instructions following the context synchronizing instruction that were performed using the old contents of the PTE will be discarded, with the result that these address translations will be performed again and, if there is no corresponding entry in any TLB, SLB, page walk cache, cache of Partition or Process Table entries, or implementation-specific address translation lookaside information, will use the value stored (or a value stored subsequently).

The ptesync instruction also ensures that all storage accesses associated with instructions preceding the ptesync instruction, and all Reference and Change bit updates associated with additional address translations that were performed, by the thread executing the ptesync instruction, before the ptesync instruction is executed, will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the ptesync instruction are performed with respect to that thread or mechanism.

6. There are additional software synchronization requirements for this instruction in multi-threaded environments (e.g., it may be necessary to invalidate one or more TLB entries on all threads in the system and to be able to determine that the invali-
versions have completed and that all side effects of the invalidations have taken effect).

Section 6.10 gives examples of using *tlbie*, *Store*, and related instructions to maintain the Page Table, in both multi-threaded environments and environments consisting of only a single-threaded processor.

--- Programming Note ---

In a multi-threaded system, if software locking is used to help ensure that the requirements described in Section 6.10 are satisfied, the *isync* instruction near the end of the lock acquisition sequence (see Section B.2.1.1 of Book II) may naturally provide the context synchronization that is required before the alteration.

7. The alteration must not cause an implicit branch in effective address space. Thus, when changing MSR_SF from 1 to 0, the *mtmsrd* instruction must have an effective address that is less than $2^{32} - 4$. Furthermore, when changing MSR_SF from 0 to 1, the *mtmsrd* instruction must not be at effective address $2^{32} - 4$ (see Section 6.3.2 on page 1177).

8. The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be independent of whether the alteration has taken effect.

--- Programming Note ---

If it is desired to set MSR_{IR} to 1 early in an operating system interrupt handler, advantage can sometimes be taken of the fact that EA_{0,3} are ignored when forming the real address when address translation is disabled and MSR_{AV} = 0. For example, if address translation resources are set such that effective address 0x0000_0000_0000_0000 maps to real address 0x0000_0000_0000_0000 when address translation is enabled, where n is an arbitrary 4-bit value, the following code sequence, in real page 0, can be used early in the interrupt handler.

```
  la   rx,target
  li   ry,0xn000
  sldi ry,ry,48
  or   rx,rx,ry  # set high-order nibble of target addr to 0xn
  mtctr rx
  bcctr # branch to target
```

targ:  mfsr  rx
      orir x,rx,0x0020
      mtmsrd rx  # set MSR_{IR} to 1

The *mtmsrd* does not cause an implicit branch in real address space because the real address of the next sequential instruction is independent of MSR_{IR}. Using *mtmsrd*, rather than *rfid* (or similar context synchronizing instruction that alters the control flow), may yield better performance on some implementations.

(Variations on the technique are possible. For example, the target instruction of the *bcctr* can be in arbitrary real page P, where P is a 48-bit value, provided that effective address 0xn || P || 0x000 maps to real address P || 0x000 when address translation is enabled.)

9. The elapsed time between the contents of the Decrementer or Hypervisor Decrementer becoming negative and the signaling of the corresponding exception is not defined.

10. If an *slbme* instruction alters the mapping, or associated attributes, of a currently mapped ESID, the *slbme* must be preceded by an *slbie* (or *slbia*) instruction that invalidates the existing translation. This applies even if the corresponding entry is no longer in the SLB (the translation may still be in implementation-specific address translation lookaside information). No software synchronization is needed between the *slbie* and the *slbme*, regardless of whether the index of the SLB entry (if any) containing the current translation is the same as the SLB index specified by the *slbme*.
No `%slbme` is needed if the `%slbme` instruction replaces a valid SLB entry with a mapping of a different ESID (e.g., to satisfy an SLB miss). However, the `%slbie` is needed later if and when the translation that was contained in the replaced SLB entry is to be invalidated.

11. When the URMOR or the HRMOR is modified, software must invalidate all implementation-specific lookaside information used in address translation that depends on the old contents of the register (i.e., the contents immediately before the modification). The `%slbie` instruction can be used to invalidate all such implementation-specific lookaside information.

12. A context synchronizing instruction or event that is executed or occurs when LPCR MER = 1 does not necessarily ensure that the exception effects of LPCRMER are consistent with the contents of LPCRMER. See Section 2.2.

13. This line applies regardless of which SPR number (13 or 29) is used for the AMR.

14. LPIDR when using HPT translation and LPCRHR must not be altered when MSRDR=1 or MSRIR=1; if they are, the results are undefined.

--- Programming Note ---

For instruction fetch, the prohibitions above are because of the difficulty of avoiding an implicit branch relative to the value of enabling software to avoid using hypervisor real addressing mode for the operation. For data access, the prohibitions above are to avoid errant (wrongly timed and/or for an incorrect context) speculative translation in support of hardware data prefetching. (The tables used for translation are determined by the partition ID and LPCRHR is used as a shortcut. See Section 6.7.6 for details.)

15. This line applies to the following Performance Monitor SPRs: PMC1-6, MMCR0, MMCR1, MMCR2, and MMCRA.

16. This line applies to all SPR numbers that access the BESCR (800-803, 806).

17. There are additional software synchronization requirements when an `%mtspr` instruction modifies this SPR in a multi-threaded environment. See Section 2.7.

18. As an alternative to a CSI, the execution of an `%rfebb` instruction or the occurrence of an event-based branch is sufficient to provide the necessary synchronization.

19. When LPCR ISL or PATE PS is modified, software must invalidate all implementation-specific lookaside information used in address translation that depends on the old contents of the field (i.e., the contents immediately before the modification). The `%slbie` instruction can be used to invalidate all such implementation-specific lookaside information.

20. `%hwsync` (or `%ptesync`) is required between the last instruction that causes a storage access (e.g., Load, Store, `dcbz`) for which the EA was translated using the current contents of LPIDR and/or PIDR and a subsequent `%mtlpidr` and/or `%mtpidr` that changes the LPID and/or PID value.

--- Programming Note ---

The preceding requirement permits designs to optimize `%tlbie` processing when the LPID and PID values specified by the `%tlbie` differ from those in effect on the receiving thread, by ensuring that all storage accesses that were caused by preceding instructions and for which the EA was translated using the current contents of LPIDR and PIDR have been performed with respect to all threads before the contents of these registers are changed.
Power ISA Book I-III Appendices
Appendix A. Notes on the Removal of Transactional Memory from the Architecture

Facilities that are available in problem state are generally not removed from the architecture because of the potential impact to the code base. Transactional Memory is a special case in that the major operating environments require software to check the availability of the facility prior to using it. In addition, the requirement that every transaction have a failure handler means that any application that uses TM will continue to function on a degenerate TM implementation that simply fails each transaction on the instruction that follows \texttt{tbegm}. These two facts enable a minimally-disruptive ecosystem transition to remove TM, even while continuing to provide product features such as live partition migration.

The anticipated system support for the removal of TM uses a \textit{synthetic} TM implementation when the thread is running in “Version 3.0 mode” (PCR\textsubscript{3.0} v2=0b10) or in “Version 2.07 mode” (PCR\textsubscript{3.0} v2=0b11). (See Section 2.5 of Book III.) For partitions running in “Version 3.1 mode” (PCR\textsubscript{3.0} v2=0b00), hypervisor software disables TM by setting HFSCR\textsubscript{58} (formerly the Transactional Memory Facility bit) to zero, and then ensures that the thread will always be in Non-transactional state by setting MSR\textsubscript{29:30} (formerly the Transaction State field) to 0b00 before dispatching the partition. (The latter requirement will be satisfied naturally provided that the hypervisor executes \texttt{treclaim}, as part of context switch, as required in Versions 2.07B and 3.0C of the architecture.)

Synthetic TM is required in order to provide unrestricted live migration of partitions running in PCR modes that supported TM (PCR\textsubscript{3.0}=1). The TM architecture allowed for access to TM SPRs and execution of TM instructions even in Non-transactional state, making pure, interrupt-driven emulation too great a performance risk. Instead, processor implementations provide default behaviors for TM-related operations. Each new transaction fails on the instruction after \texttt{tbegm}, going immediately to the failure handler. (Failure recording for this special case sets the Failure Persistent bit in TEXASR to reduce the likelihood that software will pointlessly retry the transaction, and also sets the Implementation-specific cause bit in TEXASR.) \texttt{mtspr} and \texttt{mspr} for TM SPRs have their normal behavior. Most of the other TM instructions behave as they would have behaved in Non-transactional state -- i.e., just changing CR0 and FXCC. (See Book II Section 5.5 and Book III Section 5.4.4 in Version 3.0C for background on the TM instructions.) These are the behaviors that will be seen by most software. Thus, under synthetic TM, a thread is never productively in Transactional state, and is in Suspended state (actually, synthetic Suspended state) only if the program was migrated, while in Suspended state, from a processor that implements Version 3.0C or Version 2.07B. The handling for applications migrated while in Suspended state is beyond the scope of this explanation. See the implementation’s user manual for details.

The architecture proper portrays TM to be fully removed. The HFSCR is used to disable TM in Version 3.0 mode because it is impractical to give the appearance that TM has been entirely removed. When the thread is in problem state or privileged non-hypervisor state, the deviations from the behavior for complete removal will be what one would expect if TM was implemented but disabled by the HFSCR. The following sections describe deviations in Version 3.1 mode with HFSCR\textsubscript{58} set to zero from the behavior if TM was completely removed. The description is somewhat redundant in describing deviations both from the cause and effect points of view. It does not discuss deviations that can happen only in hypervisor state. The hypervisor and ultravisor must not use any resource formerly allocated to TM other than HFSCR\textsubscript{58}.

A.1 Attempted Execution of TM Instructions

Non-privileged TM instructions were encoded using primary opcode 31 with extended opcodes 654, 686, 718, 750, 782, 814, 846, 878, and 910. Privileged TM instructions were encoded using primary opcode 31 with extended opcodes 942 and 1006.

An attempt in privileged non-hypervisor state to execute a TM instruction, or an attempt in problem state to execute a non-privileged TM instruction, will cause a Hypervisor Facility Unavailable interrupt with HFSCR\textsubscript{IC} = 0x05 indicating that the cause was an attempt to use a TM resource. Since the architecture portrays the instructions as reserved, a Hypervisor Emulation Assistance interrupt would be expected. The hypervisor's
A.3 Occurrence of the Hypervisor Facility Unavailable Interrupt with \(\text{HFSCR}_{\text{IC}} = 0x05\)

A Hypervisor Facility Unavailable interrupt that set \(\text{HFSCR}_{\text{IC}}\) to 0x05 indicated an attempt to execute a TM instruction or to access a TM SPR when \(\text{HFSCR}_{38}\) is set to zero. Despite that the architecture no longer includes Transactional Memory, this variant of the Hypervisor Facility Unavailable interrupt will occur in the circumstances described in the previous two sections and should be handled as described there.

A.4 Occurrence of the TM Bad Thing Type Program Interrupt

A Program interrupt that set \(\text{SRR1}_{42}\) to 1 indicated a TM Bad Thing type Program interrupt. Despite that the architecture no longer includes Transactional Memory, this variant of Program interrupt will occur as the result of an attempt to set \(\text{MSR}_{29:30}\) to a value other than 0b00 via an \(\text{mmsrd}\) or a “return from”-type instruction, including \(\text{rfebb}\). These bits had indicated the Transaction State. As they are now reserved, the Program interrupt would be unanticipated. Because the connection between the BESCR and the MSR bits is especially hard to make without referencing previous versions of the architecture, the forms of \(\text{rfebb}\) that set \(\text{MSR}_{29:30}\) to a value other than 0b00 are specified to be treated as though the instruction form is invalid. If a Program interrupt occurs from problem state with \(\text{SRR1}_{42} = 1\), the Program interrupt handler should treat the offending instruction as an illegal instruction.

A.5 Failure of Performance Monitor Counters to Count

\(\text{MMCR0}_{47}\) was formerly the Freeze Counters in Non-Transactional State bit. Despite the architecture’s portrayal of this bit as reserved, a 1 value in this bit will prevent Performance Monitor Counters from counting their assigned events. Software should set \(\text{MMCR0}_{47}\) to zero.

A.6 Behavior of SPR Bits Formerly Related to TM

Aside from handling the bits formerly related to TM as described above, a good general rule is to practice read-modify-write on the containing SPRs for TM-related bits, leaving their values unchanged. Software is generally permitted to write reserved bits with the expectation, in privileged state, of reading back the written value if the bit is implemented in hardware and
with the expectation that the bit will be ignored by hardware. Capriciously setting the relevant bits in SRR1, the CTR, or directly in the MSR may have the result discussed in Section A.4. Beyond this, the deviations take the form of hardware occasionally setting bits, such as the bits that were formerly MSR\textsubscript{TM} and BESC\textsubscript{TS}, to zero.
Appendix B. Illegal Instructions

With the exception of the instruction consisting entirely of binary 0s, the instructions in this class are available for future extensions of the Power ISA; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

The following primary opcodes are illegal.

5

The following primary opcode is used for an instruction prefix.

1

The following primary opcodes have unused extended opcodes. Their unused extended opcodes can be determined from the opcode maps in Appendix D of Book Appendices. All unused extended opcodes are illegal.

4, 6, 19, 30, 3156, 58, 59, 60, 62, 63

The following primary+extended opcodes have unused expanded opcodes. Their unused expanded opcodes can be determined from the opcode maps in Appendix C of Book Appendices. All unused expanded opcodes are illegal.

primary / extended opcode

4 / 0b10110_000001
4 / 0b11110_000001
4 / 0b11000_000010
60 / 0b01011_01000.
60 / 0b10101_1011.. 
60 / 0b11101_1011.. 
63 / 0b10010_00100.
63 / 0b11010_00100.
63 / 0b10010_00111.

An instruction consisting entirely of binary 0s is illegal, and is guaranteed to be illegal in all future versions of this architecture.
Appendix C. Reserved Instructions

The instructions in this class are allocated to specific purposes that are outside the scope of the Power ISA.

The following types of instruction are included in this class.

1. The instruction having primary opcode 0, except the instruction consisting entirely of binary 0s (which is an illegal instruction; see Section 1.7.2, "Illegal Instruction Class" on page 24) and the extended opcode shown below.

   **256**  Service Processor "Attention"

2. Instructions from the POWER Architecture that have not been included in the Power ISA.

3. Instructions defined in a previous version of the Power ISA that have been removed.

4. Implementation-specific instructions used to conform to the Power ISA specification.

5. Any other implementation-dependent instructions that are not defined in the Power ISA.
This appendix contains opcode maps showing the primary opcodes, extended opcodes, and expanded opcodes. Table 9 describes the conventions used in the opcode maps.

Table 9: Opcode Maps Legend

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>version</th>
<th>privilege</th>
<th>format</th>
</tr>
</thead>
<tbody>
<tr>
<td>po</td>
<td></td>
<td></td>
<td>po</td>
</tr>
<tr>
<td>xop</td>
<td></td>
<td></td>
<td>xop</td>
</tr>
</tbody>
</table>

- **po** primary opcode (decimal format)
- **xop** extended or expanded opcode image (binary format)
  - 0 instruction bit corresponding to an extended/expanded opcode bit having value of 0
  - 1 instruction bit corresponding to an extended/expanded opcode bit having value of 1
  - reserved instruction bit, must have value of 0, otherwise invalid form
  - . instruction bit corresponding to an operand or control bit, can have a value of either 0 or 1

**book**
- Book instruction defined

**version**
- ISA version instruction introduced

**privilege**
- P privileged instruction
- HV hypervisor-privileged instruction
- UV ultravisor-privileged instruction

**format**
- instruction format

**Illegal opcode**
- Opcode having no previous or current assignment, available for future use

**Defined opcode (primary, extended, or expanded)**
- Opcode assigned to a defined instruction

**Primary opcode having an extended opcode field**
- Opcode having extended opcode field used to identify multiple instructions

**Extended opcode having an expanded opcode field**
- Opcode having expanded opcode field used to identify multiple instructions

**Reserved opcode (primary, extended, or expanded)**
- Opcode is not available for future use without careful consideration
  1. Opcode corresponds to an instruction defined in a previous version of the architecture that has been subsequently removed from the architecture. The opcode is treated as an illegal opcode.
  2. Or, opcode is reserved for implementation-dependent use.
- These opcodes will not be assigned a meaning in the Power ISA except after careful consideration of the effect of such assignment on existing implementations.

**Invalid form opcode**
- Opcode corresponding to a defined instruction encoding with one or more reserved opcode bits having a value of 1
Table 10: Primary Opcode Map for Opcode Space 0 (32-bit instruction encoding) (bits 0-5)
Primary opcodes of word instructions are mapped to opcode space 0
Primary opcodes of suffixes of M[M](LS)RR-form prefixed instructions are mapped to opcode space 0

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>subfc</td>
<td>cmpli</td>
<td>cmpli</td>
<td>addic</td>
<td>addic.</td>
<td>[p]addi</td>
<td>[p]addis</td>
<td>000</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
<td>ori</td>
<td>xori</td>
<td>xoris</td>
<td>andi.</td>
<td>andis.</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 11: Primary Opcode Map for Opcode Space 1 (64-bit instruction encoding) (suffix bits 0-5)
Primary opcodes of suffixes of 8[M][LS]RR-form prefixed instructions are mapped to opcode space 1

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
<td>lq</td>
<td>EXT057</td>
<td>EXT058</td>
<td>EXT059</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

Table 12: PREFIX: Opcode Map (64-bit instruction encoding) (prefix bits 6:11)

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

1350 Power ISA™ Appendices
Table 13: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (Sheet 1 of 8)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vaddubm</th>
<th>vmul10cuq</th>
<th>vmxub</th>
<th>vxavgub</th>
<th>vxbarb</th>
<th>vxavgd</th>
<th>vcmpnequb()</th>
<th>vcmpneqwd()</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vadduhm</th>
<th>vmul10cuq</th>
<th>vmxuh</th>
<th>vxavguh</th>
<th>vxbarh</th>
<th>vxavgd</th>
<th>vcmpnequh()</th>
<th>vcmpneqwd()</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vaddum</th>
<th>vmxum</th>
<th>vxavgum</th>
<th>vxbarum</th>
<th>vxavgd</th>
<th>vcmpnequ()</th>
<th>vcmpneqwd()</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vaddum</th>
<th>vcmppaq</th>
<th>vmaxxu</th>
<th>vxavgxu</th>
<th>vxbaru</th>
<th>vxavgd</th>
<th>vcmpneqpx()</th>
<th>vcmpneqpwd()</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vaddwu</th>
<th>vxavguw</th>
<th>vxbaruw</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vaddwd</th>
<th>vxavgwd</th>
<th>vxbarwd</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vaddws</th>
<th>vxavgws</th>
<th>vxbarws</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>01110</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vxavgv</th>
<th>vxbarv</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vxavgv</th>
<th>vxbarv</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vxavgv</th>
<th>vxbarv</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>10010</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vxavgv</th>
<th>vxbarv</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vxavgv</th>
<th>vxbarv</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vxavgv</th>
<th>vxbarv</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>10110</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vxavgv</th>
<th>vxbarv</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vxavgv</th>
<th>vxbarv</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>11001</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vxavgv</th>
<th>vxbarv</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>11100</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>vxavgv</th>
<th>vxbarv</th>
<th>vxavgd</th>
<th>vcmpnew()</th>
<th>vcmpneqew()</th>
</tr>
</thead>
<tbody>
<tr>
<td>11101</td>
<td>000000</td>
<td>000001</td>
<td>000010</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

---

Appendix D. Opcode Maps 1351
Table 13: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (Sheet 2 of 8)

<table>
<thead>
<tr>
<th>00000</th>
<th>00100</th>
<th>01010</th>
<th>01110</th>
<th>00001</th>
<th>00101</th>
<th>01011</th>
<th>01111</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmuloub</td>
<td>vaddfp</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>XPND004-4A</td>
</tr>
<tr>
<td>vmuloub</td>
<td>vsdfp</td>
<td>vaddfr</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
</tr>
<tr>
<td>00010</td>
<td>00101</td>
<td>01010</td>
<td>01110</td>
<td>00011</td>
<td>00101</td>
<td>01011</td>
<td>01111</td>
</tr>
<tr>
<td>vmuluw</td>
<td>vmulum</td>
<td>vaddfr</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
</tr>
<tr>
<td>00011</td>
<td>00101</td>
<td>01010</td>
<td>01110</td>
<td>00011</td>
<td>00101</td>
<td>01011</td>
<td>01111</td>
</tr>
<tr>
<td>vaddfp</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
</tr>
<tr>
<td>vaddfr</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>01000</td>
<td>01001</td>
<td>01011</td>
<td>01010</td>
<td>01000</td>
<td>01001</td>
<td>01011</td>
</tr>
<tr>
<td>vaddfp</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
</tr>
<tr>
<td>vaddfr</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td>01111</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td>01100</td>
<td>01101</td>
</tr>
<tr>
<td>vaddfp</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
</tr>
<tr>
<td>vaddfr</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td>10011</td>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
<td>10000</td>
<td>10001</td>
</tr>
<tr>
<td>vaddfp</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>10101</td>
<td>10110</td>
<td>10111</td>
<td>10100</td>
<td>10101</td>
<td>10110</td>
<td>10111</td>
</tr>
<tr>
<td>vaddfr</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td>10111</td>
<td>10100</td>
<td>10101</td>
<td>10110</td>
<td>10111</td>
<td>10100</td>
<td>10101</td>
</tr>
<tr>
<td>vaddfp</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
</tr>
<tr>
<td>vaddfr</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td>11011</td>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
<td>11000</td>
<td>11001</td>
</tr>
<tr>
<td>vaddfp</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
</tr>
<tr>
<td>vaddfr</td>
<td>vdivuq</td>
<td>vspfibl</td>
<td>vextactub</td>
<td>vpkuhdp</td>
<td>vinsdbx</td>
<td>xpnd004-4b</td>
<td></td>
</tr>
</tbody>
</table>
Table 13: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (Sheet 3 of 8)
### Table 13: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (Sheet 4 of 8)

<table>
<thead>
<tr>
<th></th>
<th>011000</th>
<th>011001</th>
<th>011010</th>
<th>011011</th>
<th>011100</th>
<th>011101</th>
<th>011110</th>
<th>011111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>vextdubvix</td>
<td>vextdubvrx</td>
<td>vextduhvix</td>
<td>vextduhvrx</td>
<td>vextduwvlx</td>
<td>vextduwvrx</td>
<td>vextddvlx</td>
<td>vextddvrx</td>
</tr>
<tr>
<td>00001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00005</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 13: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (Sheet 5 of 8)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>Version</th>
<th>Accessible</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>vmhaddhs</td>
<td>v2.03</td>
<td>VA</td>
</tr>
<tr>
<td>00001</td>
<td>vmhraddhs</td>
<td>v2.03</td>
<td>VA</td>
</tr>
<tr>
<td>00010</td>
<td>vmladduhm</td>
<td>v2.03</td>
<td>VA</td>
</tr>
<tr>
<td>00011</td>
<td>vmsumudm</td>
<td>v3.0B</td>
<td>VA</td>
</tr>
<tr>
<td>00100</td>
<td>vmsumubm</td>
<td>v2.03</td>
<td>VA</td>
</tr>
<tr>
<td>00101</td>
<td>vmsummbm</td>
<td>v2.03</td>
<td>VA</td>
</tr>
<tr>
<td>00110</td>
<td>vmsumuhm</td>
<td>v2.03</td>
<td>VA</td>
</tr>
<tr>
<td>00111</td>
<td>vmsumuhs</td>
<td>v2.03</td>
<td>VA</td>
</tr>
</tbody>
</table>

Note: The table continues with more entries, but they are not shown here.
<table>
<thead>
<tr>
<th></th>
<th>101000</th>
<th>101001</th>
<th>101010</th>
<th>101011</th>
<th>101100</th>
<th>101101</th>
<th>101110</th>
<th>101111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>vmsumshm</td>
<td>vmsumshs</td>
<td>vsel</td>
<td>vperm</td>
<td>vsldoi</td>
<td>vpermxor</td>
<td>vmaddfp</td>
<td>vnmsubfp</td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 13: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (Sheet 6 of 8)
Table 13: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (Sheet 7 of 8)
Table 13: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (Sheet 8 of 8)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>vpermr</td>
<td>0000</td>
<td>vpermr</td>
<td>0000</td>
<td>vpermr</td>
<td>0000</td>
<td>vpermr</td>
<td>0000</td>
<td>vpermr</td>
<td>0000</td>
<td>vpermr</td>
<td>0000</td>
<td>vpermr</td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td>0001</td>
<td></td>
<td>0001</td>
<td></td>
<td>0001</td>
<td></td>
<td>0001</td>
<td></td>
<td>0001</td>
<td></td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td>0010</td>
<td></td>
<td>0010</td>
<td></td>
<td>0010</td>
<td></td>
<td>0010</td>
<td></td>
<td>0010</td>
<td></td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td>0011</td>
<td></td>
<td>0011</td>
<td></td>
<td>0011</td>
<td></td>
<td>0011</td>
<td></td>
<td>0011</td>
<td></td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td>0100</td>
<td></td>
<td>0100</td>
<td></td>
<td>0100</td>
<td></td>
<td>0100</td>
<td></td>
<td>0100</td>
<td></td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td>0101</td>
<td></td>
<td>0101</td>
<td></td>
<td>0101</td>
<td></td>
<td>0101</td>
<td></td>
<td>0101</td>
<td></td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td>0110</td>
<td></td>
<td>0110</td>
<td></td>
<td>0110</td>
<td></td>
<td>0110</td>
<td></td>
<td>0110</td>
<td></td>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td>0111</td>
<td></td>
<td>0111</td>
<td></td>
<td>0111</td>
<td></td>
<td>0111</td>
<td></td>
<td>0111</td>
<td></td>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>1000</td>
<td></td>
<td>1000</td>
<td></td>
<td>1000</td>
<td></td>
<td>1000</td>
<td></td>
<td>1000</td>
<td></td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td>1001</td>
<td></td>
<td>1001</td>
<td></td>
<td>1001</td>
<td></td>
<td>1001</td>
<td></td>
<td>1001</td>
<td></td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td>1010</td>
<td></td>
<td>1010</td>
<td></td>
<td>1010</td>
<td></td>
<td>1010</td>
<td></td>
<td>1010</td>
<td></td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td>1100</td>
<td></td>
<td>1100</td>
<td></td>
<td>1100</td>
<td></td>
<td>1100</td>
<td></td>
<td>1100</td>
<td></td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td>1101</td>
<td></td>
<td>1101</td>
<td></td>
<td>1101</td>
<td></td>
<td>1101</td>
<td></td>
<td>1101</td>
<td></td>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td>1110</td>
<td></td>
<td>1110</td>
<td></td>
<td>1110</td>
<td></td>
<td>1110</td>
<td></td>
<td>1110</td>
<td></td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td>1111</td>
<td></td>
<td>1111</td>
<td></td>
<td>1111</td>
<td></td>
<td>1111</td>
<td></td>
<td>1111</td>
<td></td>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>
### Table 14: EXT006: Extended Opcode Map for Opcode Space 0, Primary Opcode 6 (bits 28:31)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>lxvp</td>
<td>stxvp</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 15: EXT017: Extended Opcode Map for Opcode Space 0, Primary Opcode 17 (bits 30:31)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>scv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>sc</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 16: EXT030: Extended Opcode Map for Opcode Space 0, Primary Opcode 30 (bits 27:30)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>rldicl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>rldicr</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>rldic</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 17: EXT057: Extended Opcode Map for Opcode Space 0, Primary Opcode 57 (bits 30:31)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>lfdp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>lxsd</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>lxssp</td>
<td></td>
</tr>
</tbody>
</table>

### Table 18: EXT058: Extended Opcode Map for Opcode Space 0, Primary Opcode 58 (bits 30:31)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ld</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>ldu</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>lwa</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 19: EXT061: Extended Opcode Map for Opcode Space 0, Primary Opcode 61 (bits 29:31)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>stfdp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>lxv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>stxsd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>stxssp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 20: EXT062: Extended Opcode Map for Opcode Space 0, Primary Opcode 62 (bits 30:31)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>std</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>stdu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>stq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 21: EXT019: Extended Opcode Map for Opcode Space 0, Primary Opcode 19 (bits 28:31) (Sheet 1 of 4)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>mcrf</th>
<th>addpcis</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01000</td>
<td>01001</td>
</tr>
<tr>
<td>---</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 21: EXT019: Extended Opcode Map for Opcode Space 0, Primary Opcode 19 (bits 28:31) (Sheet 3 of 4)

<table>
<thead>
<tr>
<th></th>
<th>10000</th>
<th>10001</th>
<th>10010</th>
<th>10011</th>
<th>10100</th>
<th>10101</th>
<th>10110</th>
<th>10111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00000</td>
</tr>
<tr>
<td>00001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00001</td>
</tr>
<tr>
<td>00010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00010</td>
</tr>
<tr>
<td>00011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00011</td>
</tr>
<tr>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00100</td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00101</td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00110</td>
</tr>
<tr>
<td>00111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00111</td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01000</td>
</tr>
<tr>
<td>01001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01001</td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01010</td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01011</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01100</td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01101</td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01110</td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01111</td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10000</td>
</tr>
<tr>
<td>10001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10001</td>
</tr>
<tr>
<td>10010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10010</td>
</tr>
<tr>
<td>10011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10011</td>
</tr>
<tr>
<td>10100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10100</td>
</tr>
<tr>
<td>10101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10101</td>
</tr>
<tr>
<td>10110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10110</td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10111</td>
</tr>
<tr>
<td>11000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11000</td>
</tr>
<tr>
<td>11001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11001</td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11010</td>
</tr>
<tr>
<td>11011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11011</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11100</td>
</tr>
<tr>
<td>11101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11101</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11110</td>
</tr>
<tr>
<td>11111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11111</td>
</tr>
<tr>
<td></td>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 21: EXT019: Extended Opcode Map for Opcode Space 0, Primary Opcode 19 (bits 28:31) (Sheet 4 of 4)
Table 22: EXT031: Extended Opcode Map for Opcode Space 0, Primary Opcode 31 (bits 21:30) (Sheet 1 of 4)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>Notes</th>
<th>Opcode</th>
<th>Description</th>
<th>Notes</th>
<th>Opcode</th>
<th>Description</th>
<th>Notes</th>
<th>Opcode</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>cmp</td>
<td></td>
<td>00010</td>
<td>cmp</td>
<td></td>
<td>00100</td>
<td>cmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>cmpl</td>
<td></td>
<td>00011</td>
<td>cmpl</td>
<td></td>
<td>00101</td>
<td>cmpl</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>cmpla</td>
<td></td>
<td>00012</td>
<td>cmpla</td>
<td></td>
<td>00102</td>
<td>cmpla</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>setb</td>
<td></td>
<td>00013</td>
<td>setb</td>
<td></td>
<td>00103</td>
<td>setb</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>cmprb</td>
<td></td>
<td>00101</td>
<td>cmprb</td>
<td></td>
<td>00104</td>
<td>cmprb</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>setbcr</td>
<td></td>
<td>00102</td>
<td>setbcr</td>
<td></td>
<td>00105</td>
<td>setbcr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>cmpeqb</td>
<td></td>
<td>00111</td>
<td>cmpeqb</td>
<td></td>
<td>00112</td>
<td>cmpeqb</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>mcrxrt</td>
<td></td>
<td>10001</td>
<td>mcrxrt</td>
<td></td>
<td>10002</td>
<td>mcrxrt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td>mcrxrt</td>
<td></td>
<td>10011</td>
<td>mcrxrt</td>
<td></td>
<td>10012</td>
<td>mcrxrt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>mcrxrt</td>
<td></td>
<td>10101</td>
<td>mcrxrt</td>
<td></td>
<td>10102</td>
<td>mcrxrt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td>mcrxrt</td>
<td></td>
<td>10111</td>
<td>mcrxrt</td>
<td></td>
<td>10112</td>
<td>mcrxrt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>mcrxrt</td>
<td></td>
<td>11001</td>
<td>mcrxrt</td>
<td></td>
<td>11002</td>
<td>mcrxrt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td>mcrxrt</td>
<td></td>
<td>11011</td>
<td>mcrxrt</td>
<td></td>
<td>11012</td>
<td>mcrxrt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>mcrxrt</td>
<td></td>
<td>11101</td>
<td>mcrxrt</td>
<td></td>
<td>11102</td>
<td>mcrxrt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td>mcrxrt</td>
<td></td>
<td>11111</td>
<td>mcrxrt</td>
<td></td>
<td>11112</td>
<td>mcrxrt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1364  Power ISA™ Appendices
### Table 22: EXT031: Extended Opcode Map for Opcode Space 0, Primary Opcode 31 (bits 21:30) (Sheet 2 of 4)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>Action</th>
<th>Version</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>subc()</td>
<td>0</td>
<td></td>
<td>xsspx</td>
</tr>
<tr>
<td>00001</td>
<td>subf()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>mulhd()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>neg()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>sub()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>add()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>divde()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>subf()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>subf()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>mulhd()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>divde()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>divd()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>subf()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td>mulhd()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td>divd()</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Version 3.1**

## Appendix D. Opcode Maps
Table 22: EXT031: Extended Opcode Map for Opcode Space 0, Primary Opcode 31 (bits 21:30) (Sheet 3 of 4)

<table>
<thead>
<tr>
<th></th>
<th>10000</th>
<th>10001</th>
<th>10010</th>
<th>10011</th>
<th>10100</th>
<th>10101</th>
<th>10110</th>
<th>10111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1366  Power ISA™ Appendices
Table 22: EXT031: Extended Opcode Map for Opcode Space 0, Primary Opcode 31 (bits 21:30) (Sheet 4 of 4)

<table>
<thead>
<tr>
<th>Opcode (bits 21:30)</th>
<th>Instruction</th>
<th>Modifier</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>slw</td>
<td>I</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>cntlzw</td>
<td>P1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>sld</td>
<td>PPC</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>popcntb</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>cntlbcd</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>cbcdd</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>popcntw</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>sraw</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td>cntlz</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td>srad</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>sraw</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td>sraw</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>sraw</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td>sraw</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td>sraw</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>sraw</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td>extsw</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

*Table continues...*
Table 23: EXT059: Extended Opcode Map for Opcode Space 0, Primary Opcode 59 (bits 21:30) (Sheet 1 of 4)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
<th>00000</th>
<th>00010</th>
<th>00100</th>
<th>00101</th>
<th>00110</th>
<th>00111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
</tr>
<tr>
<td></td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
</tr>
<tr>
<td></td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>00000</td>
<td>00001</td>
</tr>
<tr>
<td></td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
<td>dadd</td>
</tr>
<tr>
<td></td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
<td>v2.05</td>
</tr>
</tbody>
</table>

(Continued on the next page...)

---

1368  Power ISA™ Appendices
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Function</th>
<th>Version 3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>pm:xvi8gerpp</td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>pm:xvi8ger4</td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>pm:xvi16ger2pp</td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>pm:xvi32gerpp</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>pm:xvi4ger8pp</td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>pm:xvi16ger2sspp</td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>pm:xvf16ger2pp</td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>pm:xvf32gerpp</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>pm:xvi16ger2p</td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>pm:xvf16ger2np</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>pm:xvf32gernp</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>pm:xvi8ger4spp</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>pm:xvi16ger2pp</td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td>pm:xvf16ger2np</td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td>pm:xvf32gernp</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>pm:xvf16ger2pn</td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td>pm:xvf16ger2nn</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>pm:xvf32gerpn</td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>pm:xvf64gernp</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>pm:xvf16ger2nn</td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td>pm:xvf32gerpn</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>pm:xvf64gerpn</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>pm:xvi8ger4</td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>pm:xvi16ger2</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>pm:xvf16ger2p</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>pm:xvf32gerp</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>pm:xvi16ger2pp</td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td>pm:xvf16ger2np</td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td>pm:xvf32gernp</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>pm:xvf16ger2pn</td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td>pm:xvf16ger2nn</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>pm:xvf32gerpn</td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>pm:xvf64gerpn</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>pm:xvf16ger2nn</td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td>pm:xvf32gerpn</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>pm:xvf64gerpn</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>pm:xvi8ger4</td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>pm:xvi16ger2</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>pm:xvf16ger2p</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>pm:xvf32gerp</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>pm:xvi16ger2pp</td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td>pm:xvf16ger2np</td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td>pm:xvf32gernp</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>pm:xvf16ger2pn</td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td>pm:xvf16ger2nn</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>pm:xvf32gerpn</td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>pm:xvf64gerpn</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>pm:xvf16ger2nn</td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td>pm:xvf32gerpn</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>pm:xvf64gerpn</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>pm:xvi8ger4</td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>pm:xvi16ger2</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>pm:xvf16ger2p</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>pm:xvf32gerp</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>pm:xvi16ger2pp</td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td>pm:xvf16ger2np</td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td>pm:xvf32gernp</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>pm:xvf16ger2pn</td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td>pm:xvf16ger2nn</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>pm:xvf32gerpn</td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>pm:xvf64gerpn</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>pm:xvf16ger2nn</td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td>pm:xvf32gerpn</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>pm:xvf64gerpn</td>
<td></td>
</tr>
</tbody>
</table>
Table 23: EXT059: Extended Opcode Map for Opcode Space 0, Primary Opcode 59 (bits 21:30) (Sheet 3 of 4)
### Table 23: EXT059: Extended Opcode Map for Opcode Space 0, Primary Opcode 59 (bits 21:30) (Sheet 4 of 4)

<table>
<thead>
<tr>
<th>11000</th>
<th>11001</th>
<th>11010</th>
<th>11011</th>
<th>11100</th>
<th>11101</th>
<th>11110</th>
<th>11111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 24: EXT060: Extended Opcode Map for Opcode Space 0, Primary Opcode 60 (bits 21:30) (Sheet 1 of 4)

<table>
<thead>
<tr>
<th></th>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
<th>00100</th>
<th>00101</th>
<th>00110</th>
<th>00111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>xsaddsp</td>
<td>xsmaddasp</td>
<td>xsmaddmsp</td>
<td>xsmaddbasp</td>
<td>xsmaddbmsp</td>
<td>xsmaddbmdp</td>
<td>xsmaddbmdp</td>
<td>xsmaddbmdp</td>
</tr>
<tr>
<td>00010</td>
<td>xsmulsp</td>
<td>xsmulmsp</td>
<td>xsmulbsp</td>
<td>xsmulbspsp</td>
<td>xsmulbspspsp</td>
<td>xsmulbspspspsp</td>
<td>xsmulbspspspspsp</td>
<td>xsmulbspspspspspsp</td>
</tr>
<tr>
<td>00100</td>
<td>xsdvsp</td>
<td>xsdvmsp</td>
<td>xsdvbsp</td>
<td>xsdvbspsp</td>
<td>xsdvbspspsp</td>
<td>xsdvbspspspsp</td>
<td>xsdvbspspspspsp</td>
<td>xsdvbspspspspspsp</td>
</tr>
<tr>
<td>00110</td>
<td>xsmuldp</td>
<td>xsmuldmsp</td>
<td>xsmuldbsp</td>
<td>xsmuldbspsp</td>
<td>xsmuldbspspsp</td>
<td>xsmuldbspspspsp</td>
<td>xsmuldbspspspspsp</td>
<td>xsmuldbspspspspspsp</td>
</tr>
<tr>
<td>01000</td>
<td>xvaddsp</td>
<td>xvaddmsp</td>
<td>xvaddbsp</td>
<td>xvaddbspsp</td>
<td>xvaddbspspsp</td>
<td>xvaddbspspspsp</td>
<td>xvaddbspspspspsp</td>
<td>xvaddbspspspspspsp</td>
</tr>
<tr>
<td>01010</td>
<td>xvmulsp</td>
<td>xvmulmsp</td>
<td>xvmulbsp</td>
<td>xvmulbspsp</td>
<td>xvmulbspspsp</td>
<td>xvmulbspspspsp</td>
<td>xvmulbspspspspsp</td>
<td>xvmulbspspspspspsp</td>
</tr>
<tr>
<td>01100</td>
<td>xvadddp</td>
<td>xvadddmsp</td>
<td>xvadddbsp</td>
<td>xvadddbspsp</td>
<td>xvadddbspspsp</td>
<td>xvadddbspspspsp</td>
<td>xvadddbspspspspsp</td>
<td>xvadddbspspspspspsp</td>
</tr>
<tr>
<td>01110</td>
<td>xvmuldp</td>
<td>xvmuldmsp</td>
<td>xvmuldbsp</td>
<td>xvmuldbspsp</td>
<td>xvmuldbspspsp</td>
<td>xvmuldbspspspsp</td>
<td>xvmuldbspspspspsp</td>
<td>xvmuldbspspspspspsp</td>
</tr>
<tr>
<td>10000</td>
<td>xsmaxcdp</td>
<td>xsmaxcdsp</td>
<td>xsmaxcdmsp</td>
<td>xsmaxcdmbsp</td>
<td>xsmaxcdmbspsp</td>
<td>xsmaxcdmbspspsp</td>
<td>xsmaxcdmbspspspsp</td>
<td>xsmaxcdmbspspspspsp</td>
</tr>
<tr>
<td>10010</td>
<td>xsmindcp</td>
<td>xsmindcsp</td>
<td>xsmindcmsp</td>
<td>xsmindcmbsp</td>
<td>xsmindcmbmsp</td>
<td>xsmindcmbmspsp</td>
<td>xsmindcmbmspspsp</td>
<td>xsmindcmbmspspspsp</td>
</tr>
<tr>
<td>10100</td>
<td>xsmnpjdp</td>
<td>xsmnpjdsp</td>
<td>xsmnpjdmsp</td>
<td>xsmnpjdbsp</td>
<td>xsmnpjdbmsp</td>
<td>xsmnpjdbmspsp</td>
<td>xsmnpjdbmspspsp</td>
<td>xsmnpjdbmspspspsp</td>
</tr>
<tr>
<td>10110</td>
<td>xsmindnp</td>
<td>xsmindnsp</td>
<td>xsmindnmsp</td>
<td>xsmindnbsp</td>
<td>xsmindnbspsp</td>
<td>xsmindnbspspsp</td>
<td>xsmindnbspspspsp</td>
<td>xsmindnbspspspspsp</td>
</tr>
<tr>
<td>11000</td>
<td>xvmmaxsp</td>
<td>xvmmaxasp</td>
<td>xvmmaxasm</td>
<td>xvmmaxasmp</td>
<td>xvmmaxasmsp</td>
<td>xvmmaxasmspsp</td>
<td>xvmmaxasmspspsp</td>
<td>xvmmaxasmspspspsp</td>
</tr>
<tr>
<td>11010</td>
<td>xvmindsp</td>
<td>xvmindasp</td>
<td>xvmindasm</td>
<td>xvmindasm</td>
<td>xvmindasm</td>
<td>xvmindasm</td>
<td>xvmindasm</td>
<td>xvmindasm</td>
</tr>
<tr>
<td>11100</td>
<td>xvcpsgnsp</td>
<td>xvcpsgnasp</td>
<td>xvcpsgnasm</td>
<td>xvcpsgnasm</td>
<td>xvcpsgnasm</td>
<td>xvcpsgnasm</td>
<td>xvcpsgnasm</td>
<td>xvcpsgnasm</td>
</tr>
</tbody>
</table>

1372 Power ISA™ Appendices
<table>
<thead>
<tr>
<th></th>
<th>01000</th>
<th>01001</th>
<th>01010</th>
<th>01011</th>
<th>01100</th>
<th>01101</th>
<th>01110</th>
<th>01111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>xxsldwi</td>
<td>0000</td>
<td>xscmpd</td>
<td>0000</td>
<td>xscmpqdp</td>
<td>0000</td>
<td>xscmpqdp</td>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
<td>xxperm</td>
<td>0001</td>
<td>xscmpgdp</td>
<td>0001</td>
<td>xscmpgdp</td>
<td>0001</td>
<td>xscmpgdp</td>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
<td>xxmrglw</td>
<td>0010</td>
<td>xxmrglw</td>
<td>0010</td>
<td>xxmrglw</td>
<td>0010</td>
<td>xxmrglw</td>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
<td>xxperm</td>
<td>0011</td>
<td>xscmpexpdp</td>
<td>0011</td>
<td>xscmpexpdp</td>
<td>0011</td>
<td>xscmpexpdp</td>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
<td>xxsldwi</td>
<td>0100</td>
<td>xxsldwi</td>
<td>0100</td>
<td>xxsldwi</td>
<td>0100</td>
<td>xxsldwi</td>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
<td>xxperm</td>
<td>0101</td>
<td>xxperm</td>
<td>0101</td>
<td>xxperm</td>
<td>0101</td>
<td>xxperm</td>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
<td>xxmrglw</td>
<td>0110</td>
<td>xxmrglw</td>
<td>0110</td>
<td>xxmrglw</td>
<td>0110</td>
<td>xxmrglw</td>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
<td>xxperm</td>
<td>0111</td>
<td>xxperm</td>
<td>0111</td>
<td>xxperm</td>
<td>0111</td>
<td>xxperm</td>
<td>0111</td>
</tr>
<tr>
<td>1000</td>
<td>xxiand</td>
<td>1000</td>
<td>xxiand</td>
<td>1000</td>
<td>xxiand</td>
<td>1000</td>
<td>xxiand</td>
<td>1000</td>
</tr>
<tr>
<td>1001</td>
<td>xxiandc</td>
<td>1001</td>
<td>xxiandc</td>
<td>1001</td>
<td>xxiandc</td>
<td>1001</td>
<td>xxiandc</td>
<td>1001</td>
</tr>
<tr>
<td>1010</td>
<td>xxlor</td>
<td>1010</td>
<td>xxlor</td>
<td>1010</td>
<td>xxlor</td>
<td>1010</td>
<td>xxlor</td>
<td>1010</td>
</tr>
<tr>
<td>1011</td>
<td>xxlor</td>
<td>1011</td>
<td>xxlor</td>
<td>1011</td>
<td>xxlor</td>
<td>1011</td>
<td>xxlor</td>
<td>1011</td>
</tr>
<tr>
<td>1100</td>
<td>xxlor</td>
<td>1100</td>
<td>xxlor</td>
<td>1100</td>
<td>xxlor</td>
<td>1100</td>
<td>xxlor</td>
<td>1100</td>
</tr>
<tr>
<td>1101</td>
<td>xxlor</td>
<td>1101</td>
<td>xxlor</td>
<td>1101</td>
<td>xxlor</td>
<td>1101</td>
<td>xxlor</td>
<td>1101</td>
</tr>
<tr>
<td>1110</td>
<td>xxlor</td>
<td>1110</td>
<td>xxlor</td>
<td>1110</td>
<td>xxlor</td>
<td>1110</td>
<td>xxlor</td>
<td>1110</td>
</tr>
<tr>
<td>1111</td>
<td>xxlor</td>
<td>1111</td>
<td>xxlor</td>
<td>1111</td>
<td>xxlor</td>
<td>1111</td>
<td>xxlor</td>
<td>1111</td>
</tr>
</tbody>
</table>

Table 24: EXT060: Extended Opcode Map for Opcode Space 0, Primary Opcode 60 (bits 21:30) (Sheet 2 of 4)
Table 24: EXT060: Extended Opcode Map for Opcode Space 0, Primary Opcode 60 (bits 21:30) (Sheet 3 of 4)

<table>
<thead>
<tr>
<th>10000</th>
<th>10001</th>
<th>10010</th>
<th>10011</th>
<th>10100</th>
<th>10101</th>
<th>10110</th>
<th>10111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>xscvdpuw</td>
<td>xscvdpuw</td>
<td>xscvdpuw</td>
<td>xscvdpuw</td>
<td>xscvdpuw</td>
<td>xscvdpuw</td>
<td>xscvdpuw</td>
</tr>
<tr>
<td>00101</td>
<td>xscvdpsw</td>
<td>xscvdpsw</td>
<td>xscvdpsw</td>
<td>xscvdpsw</td>
<td>xscvdpsw</td>
<td>xscvdpsw</td>
<td>xscvdpsw</td>
</tr>
<tr>
<td>00110</td>
<td>xsvxsw</td>
<td>xsvxsw</td>
<td>xsvxsw</td>
<td>xsvxsw</td>
<td>xsvxsw</td>
<td>xsvxsw</td>
<td>xsvxsw</td>
</tr>
<tr>
<td>00111</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
</tr>
<tr>
<td>01000</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
</tr>
<tr>
<td>01001</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
<td>xcvsvpx</td>
</tr>
<tr>
<td>01010</td>
<td>xcvuvx</td>
<td>xcvuvx</td>
<td>xcvuvx</td>
<td>xcvuvx</td>
<td>xcvuvx</td>
<td>xcvuvx</td>
<td>xcvuvx</td>
</tr>
<tr>
<td>01011</td>
<td>xcvuxw</td>
<td>xcvuxw</td>
<td>xcvuxw</td>
<td>xcvuxw</td>
<td>xcvuxw</td>
<td>xcvuxw</td>
<td>xcvuxw</td>
</tr>
<tr>
<td>01100</td>
<td>xcvdp</td>
<td>xcvdp</td>
<td>xcvdp</td>
<td>xcvdp</td>
<td>xcvdp</td>
<td>xcvdp</td>
<td>xcvdp</td>
</tr>
<tr>
<td>01101</td>
<td>xcvdpx</td>
<td>xcvdpx</td>
<td>xcvdpx</td>
<td>xcvdpx</td>
<td>xcvdpx</td>
<td>xcvdpx</td>
<td>xcvdpx</td>
</tr>
<tr>
<td>01110</td>
<td>xcvud</td>
<td>xcvud</td>
<td>xcvud</td>
<td>xcvud</td>
<td>xcvud</td>
<td>xcvud</td>
<td>xcvud</td>
</tr>
<tr>
<td>01111</td>
<td>xcvup</td>
<td>xcvup</td>
<td>xcvup</td>
<td>xcvup</td>
<td>xcvup</td>
<td>xcvup</td>
<td>xcvup</td>
</tr>
<tr>
<td>10000</td>
<td>xscvdps</td>
<td>xscvdps</td>
<td>xscvdps</td>
<td>xscvdps</td>
<td>xscvdps</td>
<td>xscvdps</td>
<td>xscvdps</td>
</tr>
<tr>
<td>10001</td>
<td>xscvdps</td>
<td>xscvdps</td>
<td>xscvdps</td>
<td>xscvdps</td>
<td>xscvdps</td>
<td>xscvdps</td>
<td>xscvdps</td>
</tr>
<tr>
<td>10010</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
</tr>
<tr>
<td>10011</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
<td>xsvxsp</td>
</tr>
<tr>
<td>10100</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
</tr>
<tr>
<td>10101</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
<td>xscvdpx</td>
</tr>
<tr>
<td>10110</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
</tr>
<tr>
<td>10111</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
</tr>
<tr>
<td>11000</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
</tr>
<tr>
<td>11001</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
</tr>
<tr>
<td>11010</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
</tr>
<tr>
<td>11011</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
</tr>
<tr>
<td>11100</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
</tr>
<tr>
<td>11101</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
</tr>
<tr>
<td>11110</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
</tr>
<tr>
<td>11111</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
<td>xcvuudp</td>
</tr>
</tbody>
</table>
Table 24: EXT060: Extended Opcode Map for Opcode Space 0, Primary Opcode 60 (bits 21:30) (Sheet 4 of 4)

<table>
<thead>
<tr>
<th>11000</th>
<th>11001</th>
<th>11010</th>
<th>11011</th>
<th>11100</th>
<th>11101</th>
<th>11110</th>
<th>11111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
</tr>
<tr>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
</tr>
<tr>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
<td>10100</td>
<td>10101</td>
<td>10110</td>
<td>10111</td>
</tr>
<tr>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
</tr>
</tbody>
</table>

**xxsel** \[x\]
Table 25: EXT063: Extended Opcode Map for Opcode Space 0, Primary Opcode 63 (bits 21:30) (Sheet 1 of 4)

<table>
<thead>
<tr>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
</tr>
<tr>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
</tr>
<tr>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
</tr>
<tr>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
</tr>
</tbody>
</table>

1376  Power ISA™ Appendices
Table 25: EXT063: Extended Opcode Map for Opcode Space 0, Primary Opcode 63 (bits 21:30) (Sheet 2 of 4)
### Table 25: EXT063: Extended Opcode Map for Opcode Space 0, Primary Opcode 63 (bits 21:30) (Sheet 3 of 4)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>fdiv[.]</th>
<th>fsub[.]</th>
<th>fadd[.]</th>
<th>fsqrt[.]</th>
<th>fsel[.]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 25: EXT063: Extended Opcode Map for Opcode Space 0, Primary Opcode 63 (bits 21:30) (Sheet 4 of 4)
### Table 26: EXT132: Extended Opcode Map for Opcode Space 1, Primary Opcode 32 (bits 21:30)

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>xxsplit32dx</td>
<td>01</td>
<td>BR R</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 27: EXT133: Extended Opcode Map for Opcode Space 1, Primary Opcode 33 (bits 26:27)

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>xxblendvb</td>
<td>01</td>
<td>BR R</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>xxblendvh</td>
<td></td>
<td></td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>xxblendv</td>
<td></td>
<td></td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 28: EXT134: Extended Opcode Map for Opcode Space 1, Primary Opcode 34 (bits 26:30)

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>xxpermx</td>
<td>01</td>
<td>BR R</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>xxeval</td>
<td></td>
<td></td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

1380  Power ISA™ Appendices
### Table 29: XPND004-1A: Expanded Opcode Map for Instruction 0x10000581 (bits 11:15)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

### Table 30: XPND004-1B: Expanded Opcode Map for Instruction 0x10000781 (bits 11:15)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

### Table 31: XPND004-2: Expanded Opcode Map for Instruction 0x10000602 (bits 11:15)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>vclzlsbb</td>
<td>vctzlsbb</td>
<td>vclzlsbb</td>
<td>vctzlsbb</td>
<td>vclzlsbb</td>
<td>vctzlsbb</td>
<td>vclzlsbb</td>
<td>vctzlsbb</td>
<td>vclzlsbb</td>
<td>vctzlsbb</td>
<td>vclzlsbb</td>
<td>vctzlsbb</td>
<td>vclzlsbb</td>
<td>vctzlsbb</td>
</tr>
</tbody>
</table>

### Table 32: XPND004-3: Expanded Opcode Map for Instruction 0x1000_0642 (bits 11:15)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
<td>vexpandbm</td>
</tr>
</tbody>
</table>

### Table 33: XPND004-4A: Expanded Opcode Map for Instruction 0x1000000D (bits 11:15)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>vstribl</td>
<td>vstribr</td>
<td>vstribl</td>
<td>vstribr</td>
<td>vstribl</td>
<td>vstribr</td>
<td>vstribl</td>
<td>vstribr</td>
<td>vstribl</td>
<td>vstribr</td>
<td>vstribl</td>
<td>vstribr</td>
<td>vstribl</td>
<td>vstribr</td>
</tr>
</tbody>
</table>
### Table 34: XPND004-4B: Expanded Opcode Map for Instruction 0x1000040D (bits 11:15)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>vstribl.</td>
<td>vstribl.</td>
<td>vstribl.</td>
<td>vstribl.</td>
<td>vstribl.</td>
<td>vstribl.</td>
<td>vstribl.</td>
<td>vstribl.</td>
</tr>
<tr>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
</tr>
</tbody>
</table>

### Table 35: XPND031: Expanded Opcode Map for Instruction 0x7C000162 (bits 11:15)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxmfacc</td>
<td>xxmtacc</td>
<td>xxsetaccz</td>
<td>xxsetaccz</td>
<td>xxsetaccz</td>
<td>xxsetaccz</td>
<td>xxsetaccz</td>
<td>xxsetaccz</td>
</tr>
<tr>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
</tr>
</tbody>
</table>

### Table 36: XPND060-1: Expanded Opcode Map for Instruction 0xF000_02D0 (bits 11:15)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxspltib</td>
<td>xxspltib</td>
<td>xxspltib</td>
<td>xxspltib</td>
<td>xxspltib</td>
<td>xxspltib</td>
<td>xxspltib</td>
<td>xxspltib</td>
</tr>
<tr>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
</tr>
</tbody>
</table>

### Table 37: XPND060-2: Expanded Opcode Map for Instruction 0xF000_056C (bits 11:15)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxexpdp</td>
<td>xxexpdp</td>
<td>xxexpdp</td>
<td>xxexpdp</td>
<td>xxexpdp</td>
<td>xxexpdp</td>
<td>xxexpdp</td>
<td>xxexpdp</td>
</tr>
<tr>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
</tr>
</tbody>
</table>

### Table 38: XPND060-3: Expanded Opcode Map for Instruction 0xF000_076C (bits 11:15)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvxexpdp</td>
<td>xvxexpdp</td>
<td>xvxexpdp</td>
<td>xvxexpdp</td>
<td>xvxexpdp</td>
<td>xvxexpdp</td>
<td>xvxexpdp</td>
<td>xvxexpdp</td>
</tr>
<tr>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
<td>vx</td>
</tr>
</tbody>
</table>
### Table 39: XPND063-1: Expanded Opcode Map for Instruction 0xF000_07C4 (bits 11:15)

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 40: XPND063-2: Expanded Opcode Map for Instruction 0xF000_0648 (bits 11:15)

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 41: XPND063-3: Expanded Opcode Map for Instruction 0xF000_0688 (bits 11:15)

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 42: XPND063-4: Expanded Opcode Map for Instruction 0xF000_048E (bits 11:15)

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Appendix E. Power ISA Instruction Set Sorted by Opcode

This appendix lists all the instructions in the Power ISA, sorted by primary opcode (bits 0-5), then by extended opcode column (bits 26:31, if any), then by extended opcode row (bits 21:25, if any), then by expanded opcode (bits 11:15, if any).

<table>
<thead>
<tr>
<th>Instruction&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Book</th>
<th>Compliancy Subsets&lt;sup&gt;2&lt;/sup&gt;</th>
<th>Linux Optional Category&lt;sup&gt;3&lt;/sup&gt;</th>
<th>Always Optional Category&lt;sup&gt;4&lt;/sup&gt;</th>
<th>Mnemonic</th>
<th>Version&lt;sup&gt;5&lt;/sup&gt;</th>
<th>Privilege&lt;sup&gt;6&lt;/sup&gt;</th>
<th>Mode Dep&lt;sup&gt;7&lt;/sup&gt;</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>tdi</td>
<td>PPC</td>
<td>PT</td>
<td></td>
<td>98</td>
<td>Trap Doubleword Immediate D-form</td>
</tr>
<tr>
<td>000011</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>twi</td>
<td>PT</td>
<td></td>
<td></td>
<td>97</td>
<td>Trap Word Immediate D-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vaddubm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>325</td>
<td>Vector Add Unsigned Byte Modulo VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vaddubm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>326</td>
<td>Vector Add Unsigned Halfword Modulo VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vaddubm</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>326</td>
<td>Vector Add Unsigned Doubleword Modulo VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vadderw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>329</td>
<td>Vector Add Signed Quadword Modulo VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vaddcuq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>330</td>
<td>Vector Add &amp; write Carry Signed Quadword VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vaddcuw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>327</td>
<td>Vector Add &amp; write Carry Signed Word VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vaddhus</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>327</td>
<td>Vector Add Signed Halfword Saturate VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vaddusw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>328</td>
<td>Vector Add Signed Word Saturate VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vaddusb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>323</td>
<td>Vector Add Signed Byte Saturate VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vaddshs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>324</td>
<td>Vector Add Signed Word Saturate VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vaddshw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>324</td>
<td>Vector Add Signed Word Saturate VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vsububm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>333</td>
<td>Vector Subtract Signed Halfword Modulo VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vsububm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>334</td>
<td>Vector Subtract Signed Halfword Modulo VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vsubusw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>334</td>
<td>Vector Subtract Signed Doubleword Modulo VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vsubudm</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>337</td>
<td>Vector Subtract Signed Quadword Modulo VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vsubucq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>338</td>
<td>Vector Subtract &amp; write Carry-out Signed Quadword VX-form</td>
</tr>
<tr>
<td>000100</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vsubucw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>331</td>
<td>Vector Subtract &amp; Write Carry-out Signed Word VX-form</td>
</tr>
<tr>
<td>000010</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vsubsbs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>335</td>
<td>Vector Subtract Signed Byte Saturate VX-form</td>
</tr>
<tr>
<td>000010</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vsubsbs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>336</td>
<td>Vector Subtract Signed Halfword Saturate VX-form</td>
</tr>
<tr>
<td>000010</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vsusbs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>332</td>
<td>Vector Subtract Signed Halfword Saturate VX-form</td>
</tr>
<tr>
<td>000010</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vsusbs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>332</td>
<td>Vector Subtract Signed Word Saturate VX-form</td>
</tr>
<tr>
<td>000010</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vmul10cuq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>467</td>
<td>Vector Multiply-by-10 &amp; write Carry-out Signed Quadword VX-form</td>
</tr>
<tr>
<td>000010</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vmul10ecuq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>486</td>
<td>Vector Multiply-by-10 Extended &amp; write Carry-out Signed Quadword VX-form</td>
</tr>
<tr>
<td>000010</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>sxx</td>
<td>vcmpuq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>401</td>
<td>Vector Compare Signed Quadword VX-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 1 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcmpsq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td>401</td>
<td></td>
</tr>
<tr>
<td>vmul10uq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>487</td>
<td></td>
</tr>
<tr>
<td>vmul10ueq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>488</td>
<td></td>
</tr>
<tr>
<td>bcdcpagn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>489</td>
<td></td>
</tr>
<tr>
<td>bcdadd.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td>478</td>
<td></td>
</tr>
<tr>
<td>bcdsub.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td>478</td>
<td></td>
</tr>
<tr>
<td>bcdd.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>491</td>
<td></td>
</tr>
<tr>
<td>bcddtrunc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>494</td>
<td></td>
</tr>
<tr>
<td>bcddtrunc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>495</td>
<td></td>
</tr>
<tr>
<td>bcdctaq.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>486</td>
<td></td>
</tr>
<tr>
<td>bcdfdq.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>485</td>
<td></td>
</tr>
<tr>
<td>bcdfun.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>484</td>
<td></td>
</tr>
<tr>
<td>bcdd.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>483</td>
<td></td>
</tr>
<tr>
<td>bcdd.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>481</td>
<td></td>
</tr>
<tr>
<td>bcddtrunc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>490</td>
<td></td>
</tr>
<tr>
<td>bcdd.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>493</td>
<td></td>
</tr>
<tr>
<td>vmaaxub.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>380</td>
<td></td>
</tr>
<tr>
<td>vmaaxuh.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>381</td>
<td></td>
</tr>
<tr>
<td>vmaaxuw.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>382</td>
<td></td>
</tr>
<tr>
<td>vmaaxud.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td>383</td>
<td></td>
</tr>
<tr>
<td>vmaax.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>380</td>
<td></td>
</tr>
<tr>
<td>vmaaxsh.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>381</td>
<td></td>
</tr>
<tr>
<td>vmaaxsw.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>382</td>
<td></td>
</tr>
<tr>
<td>vmaaxed.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td>383</td>
<td></td>
</tr>
<tr>
<td>vminub.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>384</td>
<td></td>
</tr>
<tr>
<td>vminuh.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>385</td>
<td></td>
</tr>
<tr>
<td>vminuw.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>386</td>
<td></td>
</tr>
<tr>
<td>vminud.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td>387</td>
<td></td>
</tr>
<tr>
<td>vminud.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>384</td>
<td></td>
</tr>
<tr>
<td>vminub.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>385</td>
<td></td>
</tr>
<tr>
<td>vminuh.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>386</td>
<td></td>
</tr>
<tr>
<td>vminuw.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>387</td>
<td></td>
</tr>
<tr>
<td>vminud.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td>388</td>
<td></td>
</tr>
<tr>
<td>vminud.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>389</td>
<td></td>
</tr>
<tr>
<td>vminub.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>385</td>
<td></td>
</tr>
<tr>
<td>vminuh.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>386</td>
<td></td>
</tr>
<tr>
<td>vminuw.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>387</td>
<td></td>
</tr>
<tr>
<td>vminud.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td>388</td>
<td></td>
</tr>
<tr>
<td>vminud.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>389</td>
<td></td>
</tr>
<tr>
<td>vzdlsbb.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>453</td>
<td></td>
</tr>
<tr>
<td>vzdlsbbb.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>453</td>
<td></td>
</tr>
<tr>
<td>vnlsw.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>453</td>
<td></td>
</tr>
<tr>
<td>vnlgd.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>371</td>
<td></td>
</tr>
<tr>
<td>vprlybw.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>459</td>
<td></td>
</tr>
<tr>
<td>vprlybu.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>459</td>
<td></td>
</tr>
<tr>
<td>vprlybq.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>460</td>
<td></td>
</tr>
<tr>
<td>vextsb2w.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>372</td>
<td></td>
</tr>
<tr>
<td>vextsh2w.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td>372</td>
<td></td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 2 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexitst2d</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>373</td>
<td>Vector Extend Sign Halfword To Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexitstw2d</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>372</td>
<td>Vector Extend Sign Word To Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexitst2q</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>374</td>
<td>Vector Extend Sign Doubleword to Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctzbp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>450</td>
<td>Vector Count Trailing Zeros Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctzhp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>450</td>
<td>Vector Count Trailing Zeros Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctzwp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>451</td>
<td>Vector Count Trailing Zeros Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctzd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>452</td>
<td>Vector Count Trailing Zeros Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexpandbmm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>466</td>
<td>Vector Expand Byte Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexpandbhm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>466</td>
<td>Vector Expand Halfword Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexpandbwm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>467</td>
<td>Vector Expand Word Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexpandaqm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>467</td>
<td>Vector Expand Doubleword Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexpandqgm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>468</td>
<td>Vector Expand Quadword Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vextractbmm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>471</td>
<td>Vector Extract Byte Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vextractbhm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>471</td>
<td>Vector Extract Halfword Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vextractbwm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>472</td>
<td>Vector Extract Word Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vextractdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>472</td>
<td>Vector Extract Doubleword Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vextractdmm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>473</td>
<td>Vector Extract Quadword Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvsrcbmdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>463</td>
<td>Move to VSR Byte Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvsrcbwm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>463</td>
<td>Move to VSR Halfword Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvsrcwmm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>464</td>
<td>Move to VSR Word Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvsrcrdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>464</td>
<td>Move to VSR Doubleword Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvsrcrm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>465</td>
<td>Move to VSR Quadword Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcntmbb</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>469</td>
<td>Vector Count Mask Bits Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcntmbdd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>470</td>
<td>Vector Count Mask Bits Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcntmbhh</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>470</td>
<td>Vector Count Mask Bits Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcntmbhw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>470</td>
<td>Vector Count Mask Bits Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vshasigmawv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>439</td>
<td>Vector SHA-256 Sigma Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vshasigmad</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>436</td>
<td>Vector SHA-512 Sigma Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vclzd</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>447</td>
<td>Vector Count Leading Zeros Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vclzhp</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>447</td>
<td>Vector Count Leading Zeros Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vclzwp</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>448</td>
<td>Vector Count Leading Zeros Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vclzd</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>449</td>
<td>Vector Count Leading Zeros Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vabsdub</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>378</td>
<td>Vector Absolute Difference Unsigned Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vabsdhh</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>378</td>
<td>Vector Absolute Difference Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vabsdwh</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>379</td>
<td>Vector Absolute Difference Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpopcntdb</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>457</td>
<td>Vector Population Count Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpopcntbf</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>457</td>
<td>Vector Population Count Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpopcntbw</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>458</td>
<td>Vector Population Count Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpopcntdl</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>458</td>
<td>Vector Population Count Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrlb</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>404</td>
<td>Vector Rotate Left Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrlh</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>404</td>
<td>Vector Rotate Left Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrlw</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>404</td>
<td>Vector Rotate Left Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrid</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>405</td>
<td>Vector Rotate Left Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vslb</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>413</td>
<td>Vector Shift Left Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vslh</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>413</td>
<td>Vector Shift Left Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vslw</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>414</td>
<td>Vector Shift Left Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsl</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>414</td>
<td>Vector Shift Left Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrb</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>416</td>
<td>Vector Shift Right Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrh</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>416</td>
<td>Vector Shift Right Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrw</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>417</td>
<td>Vector Shift Right Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsr</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>300</td>
<td>Vector Shift Right VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsr0b</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>300</td>
<td>Vector Shift Right VX-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 3 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrah</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>419</td>
<td>Vector Shift Right Algebraic Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsraw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>420</td>
<td>Vector Shift Right Algebraic Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrad</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>420</td>
<td>Vector Shift Right Algebraic Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vandc</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>402</td>
<td>Vector Logical AND VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vor</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical OR VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vror</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical XOR VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnr</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical NOR VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vorc</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical OR with Complement VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnand</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical NAND VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>414</td>
<td>Vector Shift Left Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mvmscr</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>498</td>
<td>Move From Vector Status and Control Register VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mvmscr</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>496</td>
<td>Move To Vector Status and Control Register VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>veqv</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical Equivalence VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>verd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>417</td>
<td>Vector Shift Right Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>varv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>302</td>
<td>Vector Shift Right Variable VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>valv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>302</td>
<td>Vector Shift Left Variable VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vclzdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>449</td>
<td>Vector Count Leading Zeros Doubleword under bit Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vclzdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>449</td>
<td>Vector Count Leading Zeros Doubleword under bit Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrtnq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>405</td>
<td>Vector Rotate Left Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrtnq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>405</td>
<td>Vector Rotate Left Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrtnm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>410</td>
<td>Vector Rotate Left Word then Mask Insert VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrtnm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>410</td>
<td>Vector Rotate Left Word then Mask Insert VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrtdm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>411</td>
<td>Vector Rotate Left Doubleword then Mask Insert VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>visk</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>415</td>
<td>Vector Shift Left Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vslq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>409</td>
<td>Vector Rotate Left Quadword then AND with Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vslq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>409</td>
<td>Vector Rotate Left Quadword then AND with Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrlwnm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>407</td>
<td>Vector Rotate Left Doubleword then AND with Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrlwnm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>407</td>
<td>Vector Rotate Left Doubleword then AND with Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>varp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>418</td>
<td>Vector Shift Right Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>varp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>418</td>
<td>Vector Shift Right Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpqub[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>388</td>
<td>Vector Compare Equal Unsigned Byte VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpquh[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>388</td>
<td>Vector Compare Equal Unsigned Halfword VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpquh[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>388</td>
<td>Vector Compare Equal Unsigned Halfword VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpqub[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>388</td>
<td>Vector Compare Equal Unsigned Word VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpqub[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>388</td>
<td>Vector Compare Equal Floating-Point VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpquf[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>430</td>
<td>Vector Compare Greater Than or Equal Floating-Point VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpquf[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>430</td>
<td>Vector Compare Greater Than or Equal Floating-Point VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpgtub[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>393</td>
<td>Vector Compare Greater Than Unsigned Byte VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpgtub[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>394</td>
<td>Vector Compare Greater Than Unsigned Halfword VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpgtub[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>395</td>
<td>Vector Compare Greater Than Unsigned Word VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpgtub[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>395</td>
<td>Vector Compare Greater Than Unsigned Word VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpgtub[]</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>423</td>
<td>Vector Compare Bounds Floating-Point VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpgtub[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>396</td>
<td>Vector Compare Not Equal Byte VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpgtub[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>399</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpgtub[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>400</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpgtub[]</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>391</td>
<td>Vector Compare Equal Unsigned Doubleword VC-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 4 of 30)

1388  Power ISA™ Appendices
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Complier/Subset</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcmpnezb[ ]</td>
<td>v3.0</td>
<td>398</td>
<td>Vector Compare Not Equal or Zero Byte VC-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vcmpnezh[ ]</td>
<td>v3.0</td>
<td>399</td>
<td>Vector Compare Not Equal or Zero Halfword VC-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vcmpnezw[ ]</td>
<td>v3.0</td>
<td>400</td>
<td>Vector Compare Not Equal or Zero Word VC-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vcmpneaq[ ]</td>
<td>v3.1</td>
<td>392</td>
<td>Vector Compare Equal Quadword VC-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vcmpgtlu[ ]</td>
<td>v3.1</td>
<td>397</td>
<td>Vector Compare Greater Than Unsigned Quadword VC-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vcmpgtlu[ ]</td>
<td>v2.07</td>
<td>396</td>
<td>Vector Compare Greater Than Unsigned Doubleword VC-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vcmpgtls[ ]</td>
<td>v3.1</td>
<td>397</td>
<td>Vector Compare Greater Than Signed Quadword VC-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vcmpgtl[ ]</td>
<td>v2.07</td>
<td>396</td>
<td>Vector Compare Greater Than Signed Doubleword VC-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulutb[ ]</td>
<td>v2.03</td>
<td>340</td>
<td>Vector Multiply Odd Unsigned Byte VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmuloub[ ]</td>
<td>v2.03</td>
<td>343</td>
<td>Vector Multiply Odd Signed Word VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulouw[ ]</td>
<td>v2.07</td>
<td>344</td>
<td>Vector Multiply Odd Signed Halfword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmuloud[ ]</td>
<td>v3.1</td>
<td>345</td>
<td>Vector Multiply Odd Signed Doubleword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulust[ ]</td>
<td>v2.03</td>
<td>339</td>
<td>Vector Multiply Odd Signed Byte VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulush[ ]</td>
<td>v2.03</td>
<td>341</td>
<td>Vector Multiply Odd Signed Halfword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulousw[ ]</td>
<td>v2.07</td>
<td>343</td>
<td>Vector Multiply Odd Signed Word VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulouad[ ]</td>
<td>v3.1</td>
<td>346</td>
<td>Vector Multiply Odd Signed Doubleword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmuleub[ ]</td>
<td>v2.03</td>
<td>340</td>
<td>Vector Multiply Even Signed Byte VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmuleuh[ ]</td>
<td>v2.03</td>
<td>342</td>
<td>Vector Multiply Even Signed Halfword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmuleuw[ ]</td>
<td>v2.07</td>
<td>344</td>
<td>Vector Multiply Even Signed Word VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmuleuv[ ]</td>
<td>v3.1</td>
<td>345</td>
<td>Vector Multiply Even Signed Doubleword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulesb[ ]</td>
<td>v2.03</td>
<td>339</td>
<td>Vector Multiply Even Signed Byte VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulesh[ ]</td>
<td>v2.03</td>
<td>341</td>
<td>Vector Multiply Even Signed Halfword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulesw[ ]</td>
<td>v2.07</td>
<td>343</td>
<td>Vector Multiply Even Signed Word VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulesd[ ]</td>
<td>v3.1</td>
<td>346</td>
<td>Vector Multiply Even Signed Doubleword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vpmsumib[ ]</td>
<td>v2.07</td>
<td>440</td>
<td>Vector Polynomial Multiply-Sum Byte VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vpmsumibh[ ]</td>
<td>v2.07</td>
<td>441</td>
<td>Vector Polynomial Multiply-Sum Halfword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vpmsumim[ ]</td>
<td>v2.07</td>
<td>442</td>
<td>Vector Polynomial Multiply-Sum Word VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vpmsummiw[ ]</td>
<td>v2.07</td>
<td>443</td>
<td>Vector Polynomial Multiply-Sum Doubleword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vpmsumj[ ]</td>
<td>v2.07</td>
<td>435</td>
<td>Vector AES Cipher VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vncipher[ ]</td>
<td>v2.07</td>
<td>436</td>
<td>Vector AES Inverse Cipher VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsbox[ ]</td>
<td>v2.07</td>
<td>437</td>
<td>Vector AES SubBytes VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsun4ubs[ ]</td>
<td>v2.03</td>
<td>370</td>
<td>Vector Sum across Quarter Unsigned Byte Saturate VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsun4shs[ ]</td>
<td>v2.03</td>
<td>369</td>
<td>Vector Sum across Quarter Signed Halfword Saturate VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsun2sws[ ]</td>
<td>v2.03</td>
<td>368</td>
<td>Vector Sum across Half Signed Word Saturate VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsun4sds[ ]</td>
<td>v2.03</td>
<td>369</td>
<td>Vector Sum across Quarter Signed Byte Saturate VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsunsws[ ]</td>
<td>v2.03</td>
<td>367</td>
<td>Vector Sum across Signed Word Saturate VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmuluw[ ]</td>
<td>v2.07</td>
<td>347</td>
<td>Vector Multiply Unsigned Word Modulo VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulud[ ]</td>
<td>v3.1</td>
<td>350</td>
<td>Vector Multiply Low Doubleword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmuluih[ ]</td>
<td>v3.1</td>
<td>348</td>
<td>Vector Multiply High Unsigned Word VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmuluh[ ]</td>
<td>v3.1</td>
<td>349</td>
<td>Vector Multiply High Signed Doubleword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulhs[ ]</td>
<td>v3.1</td>
<td>347</td>
<td>Vector Multiply High Signed Word VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmulhsd[ ]</td>
<td>v3.1</td>
<td>349</td>
<td>Vector Multiply High Signed Doubleword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vapliherlast[ ]</td>
<td>v2.07</td>
<td>435</td>
<td>Vector AES Cipher Last VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vncipherlast[ ]</td>
<td>v2.07</td>
<td>436</td>
<td>Vector AES Inverse Cipher Last VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddlp[ ]</td>
<td>v2.03</td>
<td>422</td>
<td>Vector Add Floating-Point VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsublp[ ]</td>
<td>v2.03</td>
<td>422</td>
<td>Vector Subtract Floating-Point VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Book</td>
<td>Compliancy Subsets</td>
<td>Linux Optional Category</td>
<td>Always Optional Category</td>
<td>Mnemonic</td>
<td>Version</td>
<td>Privilege</td>
<td>Mode Dep</td>
<td>Page</td>
<td>Name</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>--------------------</td>
<td>------------------------</td>
<td>--------------------------</td>
<td>----------</td>
<td>---------</td>
<td>----------</td>
<td>----------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrefp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>434</td>
<td>Vector Reciprocal Estimate Floating-Point VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnsqrtfp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>434</td>
<td>Vector Reciprocal Square Root Estimate Floating-Point VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexpefp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>432</td>
<td>Vector 2 Raised to the Exponent Estimate Floating-Point VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vlogefp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>433</td>
<td>Vector Log Base 2 Estimate Floating-Point VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrfin</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>427</td>
<td>Vector Round to Floating-Point Integer Nearest VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrfiz</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>428</td>
<td>Vector Round to Floating-Point Integer toward Zero VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrfip</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>428</td>
<td>Vector Round to Floating-Point Integer toward +Infinity VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrfim</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>427</td>
<td>Vector Round to Floating-Point Integer toward -Infinity VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcfux</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>425</td>
<td>Vector Convert with round to nearest From Unsigned Word to floating-point format VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcdsx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>425</td>
<td>Vector Convert with round to nearest From Signed Word to floating-point format VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctxs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>426</td>
<td>Vector Convert with round to zero from floating-point To Signed Word format Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctxsx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>426</td>
<td>Vector Convert with round to zero from floating-point To Signed Word format Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vminfp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>424</td>
<td>Vector Maximum Floating-Point VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmnfp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>424</td>
<td>Vector Minimum Floating-Point VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvvq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>362</td>
<td>Vector Divide Unsigned Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvvv</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>358</td>
<td>Vector Divide Unsigned Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvvud</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>360</td>
<td>Vector Divide Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>362</td>
<td>Vector Divide Signed Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>358</td>
<td>Vector Divide Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsqw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>360</td>
<td>Vector Divide Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsww</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>363</td>
<td>Vector Divide Extended Unsigned Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvswwu</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>359</td>
<td>Vector Divide Extended Unsigned Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsuw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>361</td>
<td>Vector Divide Extended Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsuwu</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>361</td>
<td>Vector Divide Extended Signed Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsuwv</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>359</td>
<td>Vector Divide Extended Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsuv</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>361</td>
<td>Vector Divide Extended Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsuvu</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>361</td>
<td>Vector Divide Extended Signed Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>364</td>
<td>Vector Divide Extended Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsdv</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>365</td>
<td>Vector Divide Extended Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsdvu</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>366</td>
<td>Vector Divide Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsdvw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>366</td>
<td>Vector Divide Unsigned Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsdvx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>366</td>
<td>Vector Divide Unsigned Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsdud</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>365</td>
<td>Vector Divide Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsdqw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>366</td>
<td>Vector Divide Signed Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsdsw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>364</td>
<td>Vector Divide Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvsds</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>365</td>
<td>Vector Divide Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrmrghb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge High Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrmrghh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge High Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrmrghw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge High Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrmrghb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge Low Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrmrghh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>290</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrmrghw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>290</td>
<td>Vector Merge Low Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrsplth</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>293</td>
<td>Vector Splat Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrspltw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>294</td>
<td>Vector Splat Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrspltsb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>295</td>
<td>Vector Splat Immediate Signed Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrspltsi</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>295</td>
<td>Vector Splat Immediate Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrspltsw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>295</td>
<td>Vector Splat Immediate Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vslo</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>301</td>
<td>Vector Shift Left by Octet VX-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 6 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Compliance Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vssro</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>vsgn</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vgbdd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vgbw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vtpemrq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>vtpemrd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>vpgrow</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VX-form</td>
</tr>
<tr>
<td>Instruction</td>
<td>Book</td>
<td>Compliancy Subsets</td>
<td>Linux Optional Category</td>
<td>Always Optional Category</td>
<td>Mnemonic</td>
<td>Version</td>
<td>Privilege</td>
<td>Mode Dep</td>
<td>Page</td>
<td>Name</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>--------------------</td>
<td>-------------------------</td>
<td>--------------------------</td>
<td>----------</td>
<td>---------</td>
<td>-----------</td>
<td>----------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vupkhsb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>285</td>
<td>Vector Unpack High Signed Byte VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vupkhs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Unpack High Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vupklsb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>285</td>
<td>Vector Unpack Low Signed Byte VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vupkls</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Unpack Low Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vpkpx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>278</td>
<td>Vector Pack Pixel VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vupkhpx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>288</td>
<td>Vector Unpack High Pixel VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vupklsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>288</td>
<td>Vector Unpack Low Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vpkpx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>287</td>
<td>Vector Unpack Low Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vupkhsw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>287</td>
<td>Vector Pack Signed High Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vupklsw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>287</td>
<td>Vector Pack Signed Low Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinsbvlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>320</td>
<td>Vector Insert Byte from VSR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinshvlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>321</td>
<td>Vector Insert Halfword from VSR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinswlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>322</td>
<td>Vector Insert Word from VSR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinsw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>319</td>
<td>Vector Insert Word from GPR using immediate-specified index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinsbvrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>320</td>
<td>Vector Insert Byte from VSR using GPR-specified Right-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinshvr</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>321</td>
<td>Vector Insert Halfword from VSR using GPR-specified Right-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinswrvx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>322</td>
<td>Vector Insert Word from VSR using GPR-specified Right-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>319</td>
<td>Vector Insert Doubleword from GPR using immediate-specified index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinsblx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>315</td>
<td>Vector Insert Byte from GPR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinshlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>316</td>
<td>Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinswx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>317</td>
<td>Vector Insert Word from GPR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinsdx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>318</td>
<td>Vector Insert Doubleword from GPR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinsbrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>315</td>
<td>Vector Insert Byte from GPR using GPR-specified Right-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinshrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>316</td>
<td>Vector Insert Halfword from GPR using GPR-specified Right-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinswx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>317</td>
<td>Vector Insert Word from GPR using GPR-specified Right-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vinsdx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>318</td>
<td>Vector Insert Doubleword from GPR using GPR-specified Right-Index VX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>mtvsrbmi</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>465</td>
<td>Move To VSR Byte Mask Immediate DX-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>validdi</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>298</td>
<td>Vector Shift Left Double by Bit Immediate VN-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vaddbi</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>299</td>
<td>Vector Shift Right Double by Bit Immediate VN-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vmsumcd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>357</td>
<td>Vector Multiply-Sum &amp; write Carry-out Unsigned Doubleword VA-form</td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td>vextdubx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>309</td>
<td>Vector Extract Double Unsigned Byte to VSR using GPR-specified Left-Index VA-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 8 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Compliance Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>vextdubvrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>309</td>
<td>Vector Extract Double Unsigned Byte to VSR using GPR-specified Right-Index VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>vextduhvxlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>310</td>
<td>Vector Extract Double Unsigned Halfword to VSR using GPR-specified Left-Index VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>vextduhvrxx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>310</td>
<td>Vector Extract Double Unsigned Halfword to VSR using GPR-specified Right-Index VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vextduauxlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>311</td>
<td>Vector Extract Double Unsigned Word to VSR using GPR-specified Left-Index VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vextdauvrxx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>311</td>
<td>Vector Extract Double Unsigned Word to VSR using GPR-specified Right-Index VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vextxdvxxl</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>312</td>
<td>Vector Extract Double Doubleword to VSR using GPR-specified Left-Index VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vextxdavrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>312</td>
<td>Vector Extract Double Doubleword to VSR using GPR-specified Right-Index VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vmsuddhl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>351</td>
<td>Vector Multiply-High-Add Signed Halfword Saturate VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vmsuddhl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>351</td>
<td>Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vmsuddhm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>352</td>
<td>Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vmsumumd</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>356</td>
<td>Vector Multiply-Sum Unsigned Doubleword Modulo VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vmsumubm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>352</td>
<td>Vector Multiply-Sum Unsigned Byte Modulo VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vmsumumblm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>353</td>
<td>Vector Multiply-Sum Signed Byte Modulo VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vmsumuhm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>354</td>
<td>Vector Multiply-Sum Unsigned Halfword Modulo VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vmsumuh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>356</td>
<td>Vector Multiply-Sum Unsigned Halfword Saturate VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vmsumumsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>353</td>
<td>Vector Multiply-Sum Signed Halfword Modulo VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vmsumumsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>354</td>
<td>Vector Multiply-Sum Signed Halfword Saturate VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vsm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>297</td>
<td>Vector Select VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vperm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>296</td>
<td>Vector Permute VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vaddor</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>296</td>
<td>Vector Shift Left Double by Octet Immediate VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vpermxor</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>444</td>
<td>Vector Permute &amp; Exclusive-OR VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vmaadddp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>425</td>
<td>Vector Multiply-Add Floating-Point VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vnmssublp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>423</td>
<td>Vector Negative Multiply-Subtract Floating-Point VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>madddh</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>88</td>
<td>Multiply-Add High Doubleword VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>madddhu</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>88</td>
<td>Multiply-Add High Doubleword Unaligned VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>madddid</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>88</td>
<td>Multiply-Add Low Doubleword VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vpermr</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>296</td>
<td>Vector Permute Right-indexed VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vaddleugm v2.07</td>
<td></td>
<td></td>
<td></td>
<td>329</td>
<td>Vector Add Extended Unaligned Quadword Modulo VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vadddecqu v2.07</td>
<td></td>
<td></td>
<td></td>
<td>330</td>
<td>Vector Add Extended &amp; write Carry Unaligned Quadword Modulo VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vsudeuqmg v2.07</td>
<td></td>
<td></td>
<td></td>
<td>337</td>
<td>Vector Subtract Extended Unaligned Quadword Modulo VA-form</td>
</tr>
<tr>
<td>00100</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>vsudecqu v2.07</td>
<td></td>
<td></td>
<td></td>
<td>338</td>
<td>Vector Subtract Extended &amp; write Carry-out Unaligned Quadword VA-form</td>
</tr>
<tr>
<td>00110</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>ivkvp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>625</td>
<td>Load VSA Vector Paired DQ-form</td>
</tr>
<tr>
<td>00110</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>stvvp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>654</td>
<td>Store VSA Vector Paired DQ-form</td>
</tr>
<tr>
<td>00111</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>multip</td>
<td>P1</td>
<td></td>
<td></td>
<td>81</td>
<td>Multiply Low Immediate D-form</td>
</tr>
<tr>
<td>00110</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>subflc P1 SR</td>
<td>77</td>
<td></td>
<td></td>
<td>100</td>
<td>Subtract From Immediate Carrying D-form</td>
</tr>
<tr>
<td>00110</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>cmpli P1 SR</td>
<td>93</td>
<td></td>
<td></td>
<td>100</td>
<td>Compare Logical Immediate D-form</td>
</tr>
<tr>
<td>00110</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>cmpi P1</td>
<td>93</td>
<td></td>
<td></td>
<td>100</td>
<td>Compare Immediate D-form</td>
</tr>
<tr>
<td>00110</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>addic P1 SR</td>
<td>77</td>
<td></td>
<td></td>
<td>100</td>
<td>Add Immediate Carrying D-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 9 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110</td>
<td>LXX</td>
<td>I XXXX</td>
<td></td>
<td></td>
<td>addic.</td>
<td>P1</td>
<td>SR</td>
<td>77</td>
<td>Add Immediate Carrying and Record D-form</td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>LXX</td>
<td>I XXXX</td>
<td></td>
<td></td>
<td>addi</td>
<td>P1</td>
<td>SR</td>
<td>76</td>
<td>Add Immediate D-form</td>
<td></td>
</tr>
<tr>
<td>00000 0011</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>paddi</td>
<td>v3.1</td>
<td>CT</td>
<td>76</td>
<td>Prefixed Add Immediate MLS D-form</td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>addiss</td>
<td>P1</td>
<td>CT</td>
<td>76</td>
<td>Add Immediate Shifted D-form</td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>bit[ ] P1</td>
<td>CT</td>
<td>91</td>
<td>Move Condition Register Field XL-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>crmd</td>
<td>P1</td>
<td>CT</td>
<td>45</td>
<td>Condition Register AND with Complement XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>cmor</td>
<td>P1</td>
<td>CT</td>
<td>45</td>
<td>Condition Register NOR XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>cmnd</td>
<td>P1</td>
<td>CT</td>
<td>44</td>
<td>Condition Register AND XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>crand</td>
<td>P1</td>
<td>CT</td>
<td>44</td>
<td>Condition Register OR AND XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>crorc</td>
<td>P1</td>
<td>CT</td>
<td>44</td>
<td>Condition Register OR with Complement XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>cror</td>
<td>P1</td>
<td>CT</td>
<td>44</td>
<td>Condition Register OR XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>crol</td>
<td>P1</td>
<td>CT</td>
<td>44</td>
<td>Condition Register Logical OR XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>addpos</td>
<td>v3.0</td>
<td>CT</td>
<td>76</td>
<td>Add PC Immediate Shifted DX-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>bcd[i][ ]</td>
<td>P1</td>
<td>CT</td>
<td>42</td>
<td>Branch Conditional to Link Register XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>bccm[ ]</td>
<td>P1</td>
<td>CT</td>
<td>42</td>
<td>Branch Conditional to Count Register XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>bcmt[ ]</td>
<td>P1</td>
<td>CT</td>
<td>42</td>
<td>Branch Conditional to Target Address Register XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>rcl</td>
<td>PPC</td>
<td>P</td>
<td>1152</td>
<td>Return from Interrupt Doubleword XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>rfebb</td>
<td>v3.0</td>
<td>P</td>
<td>1152</td>
<td>Return from Event Based Branch XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>rfebb</td>
<td>v2.0</td>
<td>PPC</td>
<td>1152</td>
<td>Return from Event Based Branch Hypervisor XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>utf</td>
<td>v3.0</td>
<td>UV</td>
<td>1153</td>
<td>Ultravisor Return from Interrupt Doubleword XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>stop</td>
<td>v3.0</td>
<td>P</td>
<td>1155</td>
<td>Stop XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>isync</td>
<td>P1</td>
<td>CT</td>
<td>1076</td>
<td>Instruction Synchronize XL-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>rwinm[ ]</td>
<td>P1</td>
<td>SR</td>
<td>108</td>
<td>Rotate Left Word Immediate then Mask Insert M-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>rwinm[ ]</td>
<td>P1</td>
<td>SR</td>
<td>107</td>
<td>Rotate Left Word Immediate then AND with Mask M-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>rwinm[ ]</td>
<td>P1</td>
<td>SR</td>
<td>108</td>
<td>Rotate Left Word then AND with Mask M-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>ror</td>
<td>P1</td>
<td>CT</td>
<td>99</td>
<td>OR Immediate D-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>roris</td>
<td>P1</td>
<td>CT</td>
<td>100</td>
<td>OR Immediate Shifted D-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>xor</td>
<td>P1</td>
<td>CT</td>
<td>100</td>
<td>XOR Immediate D-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>xoris</td>
<td>P1</td>
<td>CT</td>
<td>100</td>
<td>XOR Immediate Shifted D-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>andi</td>
<td>P1</td>
<td>SR</td>
<td>99</td>
<td>AND Immediate D-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>andic</td>
<td>P1</td>
<td>SR</td>
<td>99</td>
<td>AND Immediate Shifted D-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>ricdl[ ]</td>
<td>PPC</td>
<td>SR</td>
<td>110</td>
<td>Rotate Left Doubleword Immediate then Clear Left MD-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>ricdl[ ]</td>
<td>PPC</td>
<td>SR</td>
<td>110</td>
<td>Rotate Left Doubleword Immediate then Clear Right MD-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>ricdl[ ]</td>
<td>PPC</td>
<td>SR</td>
<td>111</td>
<td>Rotate Left Doubleword Immediate then Mask Insert MD-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>rdiml[ ]</td>
<td>PPC</td>
<td>SR</td>
<td>112</td>
<td>Rotate Left Doubleword Immediate then Clear Right MDS-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>rdiml[ ]</td>
<td>PPC</td>
<td>SR</td>
<td>112</td>
<td>Rotate Left Doubleword then Clear Right MDS-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>cmpl</td>
<td>P1</td>
<td>CT</td>
<td>93</td>
<td>Compare X-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>cmpl</td>
<td>P1</td>
<td>CT</td>
<td>93</td>
<td>Compare Logical X-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>setb</td>
<td>v3.0</td>
<td>CT</td>
<td>129</td>
<td>Set Boolean X-form</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>X</td>
<td>I X</td>
<td></td>
<td></td>
<td>cmprb</td>
<td>v3.0</td>
<td>CT</td>
<td>94</td>
<td>Compare Ranged Byte X-form</td>
<td></td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 10 of 30)
Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 11 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Complimentary Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 12 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvnx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>630</td>
<td>Load VSX Vector Rightmost Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvrdx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>628</td>
<td>Load VSX Vector Rightmost Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvrtux</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>651</td>
<td>Store VSX Vector Rightmost Byte Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvrmx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>652</td>
<td>Store VSX Vector Rightmost Halfword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvnx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>652</td>
<td>Store VSX Vector Rightmost Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvrdx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>651</td>
<td>Store VSX Vector Rightmost Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>621</td>
<td>Load VSX Vector with Length X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>623</td>
<td>Load VSX Vector with Length Left-justified X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ltxvpx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>626</td>
<td>Load VSX Vector Paired Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ltxvpx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>648</td>
<td>Store VSX Vector with Length X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ltxvpx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>650</td>
<td>Store VSX Vector with Length Left-justified X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ltxvpx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>655</td>
<td>Store VSX Vector Paired Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ltxshux</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>612</td>
<td>Load VSX Scalar as Integer Byte &amp; Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ltxshux</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>612</td>
<td>Load VSX Scalar as Integer Halfword &amp; Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbxshx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>640</td>
<td>Store VSX Scalar as Integer Byte Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbxshx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>640</td>
<td>Store VSX Scalar as Integer Halfword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msqndu</td>
<td>v3.0C, UV</td>
<td>Ultraviolet Message Send X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msqndu</td>
<td>v3.0C, UV</td>
<td>Ultraviolet Message Clear X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msqndp</td>
<td>v2.07, P</td>
<td>Message Send Privileged X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msqdp</td>
<td>v2.07, P</td>
<td>Message Clear Privileged X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msqnd</td>
<td>v2.07, HV</td>
<td>Message Send X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mrbr</td>
<td>v2.07</td>
<td>Move From BHRB XFX-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cbrbrb</td>
<td>v2.07</td>
<td>Clear BHRB X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>isel</td>
<td>v2.03</td>
<td>Integer Select A-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtorf</td>
<td>P1</td>
<td>Move To Condition Register Fields X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtocrf</td>
<td>v2.01</td>
<td>Move To One Condition Register Field X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xorflacc</td>
<td>v3.1</td>
<td>VSX Move From Accumulator X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xorflacc</td>
<td>v3.1</td>
<td>VSX Move To Accumulator X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xorflacc</td>
<td>v3.1</td>
<td>VSX Set Accumulator to Zero X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtnmr</td>
<td>P1, P</td>
<td>Move To MSR X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtnsrdf</td>
<td>PPC, P</td>
<td>Move To MSR Doubleword X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tblen</td>
<td>v2.03, P</td>
<td>TLB Invalidate Entry Local X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tblie</td>
<td>P1, HV</td>
<td>TLB Invalidate Entry X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbxsync</td>
<td>v3.0, P</td>
<td>SLB Synchronize X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sibmte</td>
<td>v2.00, P</td>
<td>SLB Move To Entry X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbibie</td>
<td>PPC, P</td>
<td>SLB Invalidate Entry X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbibeg</td>
<td>v3.0, P</td>
<td>SLB Invalidate Global X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbibia</td>
<td>PPC, P</td>
<td>SLB Invalidate All X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sibagi</td>
<td>v3.0B, P</td>
<td>SLB Invalidate All Global X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtor</td>
<td>P1</td>
<td>Move From Condition Register X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtocrf</td>
<td>v2.01</td>
<td>Move From One Condition Register Field X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtnsrdf</td>
<td>v2.07</td>
<td>Move From VSX Doubleword X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtnmr</td>
<td>P1, P</td>
<td>Move From MSR X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtnsrdf</td>
<td>v2.07</td>
<td>Move To VSX Doubleword X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtovnwa</td>
<td>v2.07</td>
<td>Move To VSX Word and Zero X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtovnwa</td>
<td>v2.07</td>
<td>Move To VSX Word Algebraic X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtovnwc</td>
<td>v2.07</td>
<td>Move To VSX Word and Zero X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtovnwc</td>
<td>v2.07</td>
<td>Move To VSX Word Algebraic X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfnrdld</td>
<td>v3.0</td>
<td>Move From VSX Lower Doubleword X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 13 of 30)
## Power ISA AS Instruction Set Sorted by Opcode (Sheet 14 of 30)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compilable Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Move From Special Purpose Register XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Move From Time Base XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Move To VSR Word &amp; Splat X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Move To VSR Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Move To Special Purpose Register XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Deliver A Random Number X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SLB Move From Entry VSID X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SLB Move From Entry ESID X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SLB Find Entry ESID X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Word &amp; Reserve Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Byte And Reserve Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Doubleword And Reserve Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Halfword And Reserve Indexed Xform</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Quadword And Reserve Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Doubleword Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Doubleword Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Doubleword with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Doubleword with Update indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Word Algebraic Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Word Algebraic with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load String Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load String Word Immediate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store String Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store String Word Immediate Xform</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Word &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Halfword &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Byte &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Doubleword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Word Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Byte Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Doubleword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Word Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Doubleword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Instruction Cache Block Touch X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data Cache Block Store X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data Cache Block Flush X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data Cache Block Touch for Store X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data Cache Block Touch X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Word Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TLB Synchronize X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Synchronize X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Word Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Halfword Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Enforce In-order Execution of I/O X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Message Synchronize X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Halfword Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Instruction Cache Block Invalidate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data Cache Block set to Zero X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Word Conditional Indexed X-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 14 of 30)
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Back</th>
<th>Compliancy Subsets²</th>
<th>Linux Optional Category³</th>
<th>Always Optional Category⁴</th>
<th>Mnemonic</th>
<th>Version⁵</th>
<th>Privilege⁶</th>
<th>Mode Dep⁷</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>stycz</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Quadword Conditional Indexed X-form</td>
</tr>
<tr>
<td>stdcx</td>
<td>PPC</td>
<td>1082</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Doubleword Conditional Indexed X-form</td>
</tr>
<tr>
<td>stbcx</td>
<td>v2.06</td>
<td>1079</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Byte Conditional Indexed X-form</td>
</tr>
<tr>
<td>sthcx</td>
<td>v2.06</td>
<td>1080</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Halfword Conditional Indexed X-form</td>
</tr>
<tr>
<td>lwzx</td>
<td>P1</td>
<td>55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Word and Zero Indexed X-form</td>
</tr>
<tr>
<td>lwzux</td>
<td>P1</td>
<td>55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Word and Zero with Update Indexed X-form</td>
</tr>
<tr>
<td>lswx</td>
<td>P1</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Byte and Zero Indexed X-form</td>
</tr>
<tr>
<td>lswux</td>
<td>P1</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Byte and Zero with Update Indexed X-form</td>
</tr>
<tr>
<td>stbx</td>
<td>P1</td>
<td>61</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Word Indexed X-form</td>
</tr>
<tr>
<td>stbxu</td>
<td>P1</td>
<td>61</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Word with Update Indexed X-form</td>
</tr>
<tr>
<td>stibx</td>
<td>P1</td>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Byte Indexed X-form</td>
</tr>
<tr>
<td>stibux</td>
<td>P1</td>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Byte with Update Indexed X-form</td>
</tr>
<tr>
<td>ltdx</td>
<td>v2.05</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Halfword and Zero Indexed X-form</td>
</tr>
<tr>
<td>ltdux</td>
<td>P1</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Halfword and Zero with Update Indexed X-form</td>
</tr>
<tr>
<td>lanx</td>
<td>P1</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Halfword Algebraic Indexed X-form</td>
</tr>
<tr>
<td>lnux</td>
<td>P1</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Halfword Algebraic with Update Indexed X-form</td>
</tr>
<tr>
<td>lthx</td>
<td>P1</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Halfword Indexed X-form</td>
</tr>
<tr>
<td>lthux</td>
<td>P1</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Halfword with Update Indexed X-form</td>
</tr>
<tr>
<td>spom</td>
<td>v3.1</td>
<td>??</td>
<td>??</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Split Octword Metadata</td>
</tr>
<tr>
<td>llbx</td>
<td>P1</td>
<td>150</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Single Indexed X-form</td>
</tr>
<tr>
<td>llbux</td>
<td>P1</td>
<td>151</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Single with Update Indexed X-form</td>
</tr>
<tr>
<td>llbxu</td>
<td>P1</td>
<td>152</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Double Indexed X-form</td>
</tr>
<tr>
<td>llbdx</td>
<td>P1</td>
<td>152</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Double with Update Indexed X-form</td>
</tr>
<tr>
<td>llbsfx</td>
<td>P1</td>
<td>155</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Floating-Point Single Indexed X-form</td>
</tr>
<tr>
<td>llbsfxu</td>
<td>P1</td>
<td>156</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Floating-Point Single with Update Indexed X-form</td>
</tr>
<tr>
<td>llbfdx</td>
<td>P1</td>
<td>157</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Floating-Point Double Indexed X-form</td>
</tr>
<tr>
<td>llbfdux</td>
<td>P1</td>
<td>158</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Floating-Point Double with Update Indexed X-form</td>
</tr>
<tr>
<td>llfdp</td>
<td>v2.05</td>
<td>159</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Double Pair Indexed X-form</td>
</tr>
<tr>
<td>llfwax</td>
<td>v2.05</td>
<td>153</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point as Integer Word Algebraic Indexed X-form</td>
</tr>
<tr>
<td>llfwzx</td>
<td>v2.06</td>
<td>153</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point as Integer Word &amp; Zero Indexed X-form</td>
</tr>
<tr>
<td>stdpdx</td>
<td>v2.05</td>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Floating-Point Double Pair Indexed X-form</td>
</tr>
<tr>
<td>stfdpx</td>
<td>v2.05</td>
<td>156</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Floating-Point as Integer Word Indexed X-form</td>
</tr>
<tr>
<td>ljam</td>
<td>v3.1</td>
<td>??</td>
<td>??</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Metadata</td>
</tr>
<tr>
<td>slw[]</td>
<td>P1</td>
<td>113</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shift Left Word X-form</td>
</tr>
<tr>
<td>srr[]</td>
<td>P1</td>
<td>113</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shift Right Word X-form</td>
</tr>
<tr>
<td>sraw[]</td>
<td>P1</td>
<td>114</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shift Right Algebraic Word X-form</td>
</tr>
<tr>
<td>srawl[]</td>
<td>P1</td>
<td>114</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shift Right Algebraic Word Immediate X-form</td>
</tr>
<tr>
<td>srdat[]</td>
<td>PPC</td>
<td>115</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shift Right Algebraic Word Immediately X-form</td>
</tr>
<tr>
<td>extswl[]</td>
<td>v3.0</td>
<td>116</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Extend Sign Word and Shift Left Immediate X-form</td>
</tr>
<tr>
<td>cntizw[]</td>
<td>P1</td>
<td>102</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Count Leading Zero Word X-form</td>
</tr>
<tr>
<td>cntiz[]</td>
<td>PPC</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Count Leading Zeros Doubleword X-form</td>
</tr>
<tr>
<td>cntizcx[]</td>
<td>v3.0</td>
<td>102</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Count Trailing Zeros Word X-form</td>
</tr>
<tr>
<td>cntizdx[]</td>
<td>v3.0</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Count Trailing Zeros Doubleword X-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 15 of 30)
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Book</th>
<th>Compliancy Subsets²</th>
<th>Linux Optional Category³</th>
<th>Always Optional Category⁴</th>
<th>Mnemonic</th>
<th>Version⁵</th>
<th>Privilege⁶</th>
<th>Mode Dep⁷</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td>Shift Right Algebraic Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td>Extend Sign Halfword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td>Extend Sign Byte X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Extend Sign Word X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shift Left Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Count Leading Zeros Doubleword under bit Mask X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td>Byte-Reverse Word X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td>Byte-Reverse Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Byte-Reverse Halfword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shift Right Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Count Trailing Zeros Doubleword under bit Mask X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td>AND X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AND with Complement X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td>NOR X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NOR X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Parallel Bits Deposit Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Parallel Bits Extract Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Centrifuge Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bit Permute Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Equivalent X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td>XOR X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td>OR with Complement X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td>OR X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td>NAND X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NAND X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compare Bytes X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Wait X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Word and Zero D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Prefix Load Word and Zero MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Splat Immediate32 Doubleword Indexed 8RR:LD-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Splat Immediate Double-Precision 8RR:RD-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Splat Immediate Word 8RR:LD-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Word and Zero with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Blend Variable Byte 8RR:XX4-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Blend Variable Halfword 8RR:XX4-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Blend Variable Word 8RR:XX4-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Blend Variable Doubleword 8RR:XX4-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Byte and Zero D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Prefix Load Byte and Zero MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Permule Extended 8RR:XX4-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Evaluate 8RR:XX4-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Byte and Zero with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Store Word D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Prefixed Store Word MLS:D-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 16 of 30)

1400 Power ISA™ Appendices
<table>
<thead>
<tr>
<th>Instruction^1</th>
<th>Book</th>
<th>Compliancy Subsets^2</th>
<th>Linux Optional Category^3</th>
<th>Always Optional Category^4</th>
<th>Mnemonic</th>
<th>Version^5</th>
<th>Privilege^6</th>
<th>Mode Dep^7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stwu</td>
<td>P1</td>
<td></td>
<td></td>
<td>61</td>
<td>Store Word with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stb</td>
<td>P1</td>
<td></td>
<td></td>
<td>59</td>
<td>Store Byte D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pstb</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>59</td>
<td>Prefixed Store Byte MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stbu</td>
<td>P1</td>
<td></td>
<td></td>
<td>59</td>
<td>Store Byte with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhiz</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td>Load Halfword and Zero D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhzu</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td>Load Halfword and Zero with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plhz</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>53</td>
<td>Prefixed Load Halfword and Zero MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plwa</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>56</td>
<td>Prefixed Load Word Algebraic MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lihz</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td>Load Halfword and Zero with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plxzd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>56</td>
<td>Prefixed Load VSX Scalar Doubleword MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lihu</td>
<td>P1</td>
<td></td>
<td></td>
<td>54</td>
<td>Load Halfword Algebraic D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>liha</td>
<td>P1</td>
<td></td>
<td></td>
<td>54</td>
<td>Load Halfword Algebraic with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lihau</td>
<td>P1</td>
<td></td>
<td></td>
<td>54</td>
<td>Load Halfword Algebraic with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sthu</td>
<td>P1</td>
<td></td>
<td></td>
<td>60</td>
<td>Store Halfword D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sthw</td>
<td>P1</td>
<td></td>
<td></td>
<td>60</td>
<td>Store Halfword with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stmg</td>
<td>P1</td>
<td></td>
<td></td>
<td>60</td>
<td>Store Halfword with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pstxsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>638</td>
<td>Prefixed Store VSX Scalar Doubleword MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stmw</td>
<td>P1</td>
<td></td>
<td></td>
<td>70</td>
<td>Store Multiple Word D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pltspp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>642</td>
<td>Prefixed Store VSX Scalar Single-Precision MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>itfs</td>
<td>P1</td>
<td></td>
<td></td>
<td>150</td>
<td>Load Floating-Point Single D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>itfsf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>150</td>
<td>Prefixed Load Floating-Point Single MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>itfsu</td>
<td>P1</td>
<td></td>
<td></td>
<td>150</td>
<td>Load Floating-Point Single with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plxv</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>617</td>
<td>Prefixed Load VSX Vector MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ltd</td>
<td>P1</td>
<td></td>
<td></td>
<td>152</td>
<td>Load Floating-Point Double D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plxfd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>152</td>
<td>Prefixed Load Floating-Point Double MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ltdu</td>
<td>P1</td>
<td></td>
<td></td>
<td>152</td>
<td>Load Floating-Point Double with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stds</td>
<td>P1</td>
<td></td>
<td></td>
<td>155</td>
<td>Store Floating-Point Single D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdts</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>155</td>
<td>Prefixed Store Floating-Point Single MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdtsu</td>
<td>P1</td>
<td></td>
<td></td>
<td>155</td>
<td>Store Floating-Point Single with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pststd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>157</td>
<td>Prefixed Store Floating-Point Double MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdtdu</td>
<td>P1</td>
<td></td>
<td></td>
<td>157</td>
<td>Store Floating-Point Double with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lq</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>65</td>
<td>Load Quadword DQ-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>65</td>
<td>Prefixed Load Quadword MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pldp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>57</td>
<td>Prefixed Load Doubleword MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>llfp</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>159</td>
<td>Load Floating-Point Double Pair DS-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 17 of 30)
<table>
<thead>
<tr>
<th>Instruction(^1)</th>
<th>Book</th>
<th>Compliancy Subsets(^2)</th>
<th>Linux Optional Category(^3)</th>
<th>Always Optional Category(^4)</th>
<th>Mnemonic</th>
<th>Version(^5)</th>
<th>Privilege(^6)</th>
<th>Mode Dep(^7)</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11101 11110 ...</td>
<td>MMA</td>
<td>xxx8ger4pp</td>
<td>v3.1</td>
<td>886</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101 11110 ...</td>
<td>MMA</td>
<td>pmxx8ger4pp</td>
<td>v3.1</td>
<td>886</td>
<td>Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate MMIRR:XX3-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101 11110 ...</td>
<td>MMA</td>
<td>xvf16ger2pp</td>
<td>v3.1</td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101 11110 ...</td>
<td>MMA</td>
<td>pmxx16ger2pp</td>
<td>v3.1</td>
<td>871</td>
<td>Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101 11110 ...</td>
<td>MMA</td>
<td>xvf32gerpp</td>
<td>v3.1</td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101 11110 ...</td>
<td>MMA</td>
<td>pmxx32gerpp</td>
<td>v3.1</td>
<td>875</td>
<td>Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101 11110 ...</td>
<td>MMA</td>
<td>xx4ger8pp</td>
<td>v3.1</td>
<td>883</td>
<td>VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate MMIRR:XX3-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101 11110 ...</td>
<td>MMA</td>
<td>pmxx4ger8pp</td>
<td>v3.1</td>
<td>883</td>
<td>Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate MMIRR:XX3-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 18 of 30)
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Book</th>
<th>Compliant Subset2</th>
<th>Linux Optional Category3</th>
<th>Always Optional Category4</th>
<th>Mnemonic</th>
<th>Version5</th>
<th>Privilege6</th>
<th>Mode Dep7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>xvi16ger2spp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>893</td>
<td>VSX Vector 18-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate X3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvi16ger2spp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>893</td>
<td>Prefixed Masked VSX Vector 18-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>xvbfi6ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Positive accumulate X3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvbfi6ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>xvf64gerpp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvf64gerpp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td>Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>xvf16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate X3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvf16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>xvf32germp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvf32germp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>xvf64gerpp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>VSX Vector 64-bit Floating-Point GER (Rank-2 Update) Negative multiply, Positive accumulate X3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvf64gerpp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>Prefixed Masked VSX Vector 64-bit Floating-Point GER (Rank-2 Update) Negative multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>xvf16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td>VSX Vector 16-bit Floating-Point GER (Rank-1 update) Negative multiply, Positive accumulate X3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvf16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td>Prefixed Masked VSX Vector 16-bit Floating-Point GER (Rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>xvf64germp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvf64germp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td>Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>xvf16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate X3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvf16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>xvf32gerpn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvf32gerpn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>xvf64gerpn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (Rank-2 Update) Negative multiply, Positive accumulate X3-form</td>
</tr>
<tr>
<td>000000 1100 1000 0000 0000 0000 0000 0000</td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvf64gerpn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td>Prefixed Masked VSX Vector 64-bit Floating-Point GER (Rank-2 Update) Negative multiply, Positive accumulate MMIRR:XX3-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 19 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111111 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>00000000 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>11111111 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>00000000 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector bifloat16 GER (Rank-2 Update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>11111111 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>00000000 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>11111111 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>00000000 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>11111111 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>00000000 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>11111111 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>00000000 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>11111111 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>00000000 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>11111111 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
<tr>
<td>00000000 ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X03-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 20 of 30)
## Instruction Set Sorted by Opcode

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Compliancy Subset</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xvi16ger2pp</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>891</td>
</tr>
<tr>
<td>00000 11101</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>pmxvi16ger2pp</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>891</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>fcfids[]</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>175</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>ldivs[]</td>
<td>PPC</td>
<td>164</td>
<td>...</td>
<td>165</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>fsub[]</td>
<td>PPC</td>
<td>163</td>
<td>...</td>
<td>165</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>fadd[.]</td>
<td>PPC</td>
<td>163</td>
<td>...</td>
<td>165</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>fsubt[]</td>
<td>PPC</td>
<td>165</td>
<td>...</td>
<td>165</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>fmul[]</td>
<td>PPC</td>
<td>164</td>
<td>...</td>
<td>165</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>frsqrt[]</td>
<td>PPC</td>
<td>166</td>
<td>...</td>
<td>166</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>fmsub[]</td>
<td>PPC</td>
<td>168</td>
<td>...</td>
<td>168</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>fmaadd[]</td>
<td>PPC</td>
<td>168</td>
<td>...</td>
<td>168</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>fmsubt[]</td>
<td>PPC</td>
<td>169</td>
<td>...</td>
<td>169</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>faddd[]</td>
<td>PPC</td>
<td>169</td>
<td>...</td>
<td>169</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xaddsp</td>
<td>v2.07</td>
<td>664</td>
<td>...</td>
<td>664</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xsubsp</td>
<td>v2.07</td>
<td>811</td>
<td>...</td>
<td>811</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xsulp</td>
<td>v2.07</td>
<td>761</td>
<td>...</td>
<td>761</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xsulp</td>
<td>v2.07</td>
<td>761</td>
<td>...</td>
<td>761</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xdivsp</td>
<td>v2.07</td>
<td>721</td>
<td>...</td>
<td>721</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xaddsp</td>
<td>v2.06</td>
<td>659</td>
<td>...</td>
<td>659</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xsubsp</td>
<td>v2.06</td>
<td>807</td>
<td>...</td>
<td>807</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xmul</td>
<td>v2.06</td>
<td>757</td>
<td>...</td>
<td>757</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xdivd</td>
<td>v2.06</td>
<td>717</td>
<td>...</td>
<td>717</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xadd</td>
<td>v2.06</td>
<td>825</td>
<td>...</td>
<td>825</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xsub</td>
<td>v2.06</td>
<td>952</td>
<td>...</td>
<td>952</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xvmulp</td>
<td>v2.06</td>
<td>919</td>
<td>...</td>
<td>919</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xdivsp</td>
<td>v2.06</td>
<td>869</td>
<td>...</td>
<td>869</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xaddsp</td>
<td>v2.06</td>
<td>821</td>
<td>...</td>
<td>821</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xsubdp</td>
<td>v2.06</td>
<td>950</td>
<td>...</td>
<td>950</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xmuldp</td>
<td>v2.06</td>
<td>917</td>
<td>...</td>
<td>917</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xdivdp</td>
<td>v2.06</td>
<td>867</td>
<td>...</td>
<td>867</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xsmadd</td>
<td>v2.06</td>
<td>736</td>
<td>...</td>
<td>736</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xmmadd</td>
<td>v2.06</td>
<td>743</td>
<td>...</td>
<td>743</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xsmadd</td>
<td>v2.06</td>
<td>739</td>
<td>...</td>
<td>739</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xmmadd</td>
<td>v2.06</td>
<td>746</td>
<td>...</td>
<td>746</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xsmadd</td>
<td>v2.06</td>
<td>734</td>
<td>...</td>
<td>734</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xmmadd</td>
<td>v2.06</td>
<td>741</td>
<td>...</td>
<td>741</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xscpsgn</td>
<td>v2.06</td>
<td>682</td>
<td>...</td>
<td>682</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xvmssp</td>
<td>v2.06</td>
<td>905</td>
<td>...</td>
<td>905</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xvcpsgnsp</td>
<td>v2.06</td>
<td>838</td>
<td>...</td>
<td>838</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xvex</td>
<td>v2.06</td>
<td>896</td>
<td>...</td>
<td>896</td>
</tr>
<tr>
<td>111101 ...</td>
<td>...</td>
<td>MMA</td>
<td>MMA</td>
<td>...</td>
<td>xvmssp</td>
<td>v2.06</td>
<td>903</td>
<td>...</td>
<td>903</td>
</tr>
</tbody>
</table>

**Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 21 of 30)**
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>.000000</td>
<td>.000000</td>
<td>.000000</td>
<td>xxmindp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>907</td>
<td>VSX Vector Minimum Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000000</td>
<td>.000000</td>
<td>.000000</td>
<td>xveexpdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>896</td>
<td>VSX Vector Insert Exponent Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnaddsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>728</td>
<td>VSX Scalar Multiply-Add Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnaddmsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>728</td>
<td>VSX Scalar Multiply-Add Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnsubasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>751</td>
<td>VSX Scalar Multiply-Subtract Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnsubmsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>725</td>
<td>VSX Scalar Multiply-Add Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnmaddsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>900</td>
<td>VSX Vector Multiply-Add Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnmaddmsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>900</td>
<td>VSX Vector Multiply-Add Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnsubasp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>914</td>
<td>VSX Vector Multiply-Subtract Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnsubmsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>914</td>
<td>VSX Vector Multiply-Subtract Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnaddadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>897</td>
<td>VSX Vector Multiply-Add Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnaddmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>897</td>
<td>VSX Vector Multiply-Add Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnsubadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>911</td>
<td>VSX Vector Multiply-Subtract Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xsnsubmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>911</td>
<td>VSX Vector Multiply-Subtract Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xxnmaddsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>770</td>
<td>VSX Scalar Negative Multiply-Add Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xxnmaddmsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>770</td>
<td>VSX Scalar Negative Multiply-Add Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xxnmsubasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>779</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xxnmsubmsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>779</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xxnmaddadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>765</td>
<td>VSX Scalar Negative Multiply-Add Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xxnmaddmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>765</td>
<td>VSX Scalar Negative Multiply-Add Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xxnmsubadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>776</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xxnmsubmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>776</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xxnmaddasp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>927</td>
<td>VSX Vector Negative Multiply-Add Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xxnmaddmsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>927</td>
<td>VSX Vector Negative Multiply-Add Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.000001</td>
<td>.000001</td>
<td>.000001</td>
<td>xxnmsubasp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>933</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Single-Precision XX3-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 22 of 30)
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Book</th>
<th>Compliancy Subsets2</th>
<th>Linux Optional Category3</th>
<th>Always Optional Category4</th>
<th>Mnemonic</th>
<th>Version5</th>
<th>Privilege6</th>
<th>Mode Dep7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111101</td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td>xvmvsubsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>933</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td>xvmvaddsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>923</td>
<td>VSX Vector Negative Multiply-Add Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td>xvmvaddmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>923</td>
<td>VSX Vector Negative Multiply-Add Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td>xvmvsubadsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>930</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td>xvmvsubmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>930</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td>xxsidwi</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>990</td>
<td>VSX Vector Shift Left Double by Word Immediate XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td>xxsidwd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>990</td>
<td>VSX Vector Shift Left Double Word Immediate XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td>xxmermd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>986</td>
<td>VSX Vector Permute Doubleword Immediate XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td>xxmerghw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>982</td>
<td>VSX Vector Merge High Word XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td>xxperm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>985</td>
<td>VSX Vector Permute XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>xxmergw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>982</td>
<td>VSX Vector Merge Low Word XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>xxpermr</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>985</td>
<td>VSX Vector Permute Right-indexed XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td>xxldc</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>978</td>
<td>VSX Vector Logical AND XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td>xxldnc</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>978</td>
<td>VSX Vector Logical AND with Complement XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td>xxlor</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>981</td>
<td>VSX Vector Logical OR XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td>xxlorc</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>980</td>
<td>VSX Vector Logical OR XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td>xxlorc</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>980</td>
<td>VSX Vector Logical OR with Complement XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td>xxldand</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>979</td>
<td>VSX Vector Logical NAND XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td>xxldqv</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>979</td>
<td>VSX Vector Logical Equivalence XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td>xxldptw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>993</td>
<td>VSX Vector Split Word XX2-form</td>
</tr>
<tr>
<td>111110</td>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td>xxldptb</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>991</td>
<td>VSX Vector Split Immediate Byte X-form</td>
</tr>
<tr>
<td>111110</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>xxldtruw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>969</td>
<td>VSX Vector Extract Signed Word XX2-form</td>
</tr>
<tr>
<td>111110</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>xxldtruw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>969</td>
<td>VSX Vector Extract Signed Word XX2-form</td>
</tr>
<tr>
<td>111110</td>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td>xxldtruw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>969</td>
<td>VSX Vector Extract Signed Word XX2-form</td>
</tr>
<tr>
<td>111110</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td>xxldtruw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>969</td>
<td>VSX Vector Extract Signed Word XX2-form</td>
</tr>
<tr>
<td>111110</td>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>833</td>
<td>VSX Vector Compare Equal To Single-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>833</td>
<td>VSX Vector Compare Equal To Single-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>833</td>
<td>VSX Vector Compare Equal To Single-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>833</td>
<td>VSX Vector Compare Equal To Single-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>832</td>
<td>VSX Vector Compare Equal To Double-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>836</td>
<td>VSX Vector Compare Greater Than Double-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>834</td>
<td>VSX Vector Compare Greater Than Double-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>830</td>
<td>VSX Vector Compare Equal To Double-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>670</td>
<td>VSX Scalar Compare Equal Double-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>674</td>
<td>VSX Scalar Compare Greater Than Double-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>672</td>
<td>VSX Scalar Compare Greater Than Double-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>679</td>
<td>VSX Scalar Compare Ordered Double-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>676</td>
<td>VSX Scalar Compare Ordered Double-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td>xvmvpeqdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>668</td>
<td>VSX Scalar Compare Exponents Double-Precision XX3-form</td>
</tr>
<tr>
<td>111110</td>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td>xsvmveqdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>693</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Integer XX2-form</td>
</tr>
<tr>
<td>Instruction</td>
<td>Book</td>
<td>Compliancy Subsets</td>
<td>Linux Optional Category</td>
<td>Always Optional Category</td>
<td>Mnemonic</td>
<td>Version</td>
<td>Privilege</td>
<td>Mode Dep</td>
<td>Page</td>
<td>Name</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>--------------------</td>
<td>-------------------------</td>
<td>--------------------------</td>
<td>----------</td>
<td>---------</td>
<td>----------</td>
<td>----------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsvdpsxws</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>689</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Word format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvspuxws</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>859</td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Word format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvspxws</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>855</td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Word format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvvxwsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>866</td>
<td>VSX Vector Convert with round Unsigned Word to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvvxwsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>863</td>
<td>VSX Vector Convert with round Signed Word to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsxvuxws</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>847</td>
<td>VSX Vector Convert with round to zero Double-Precision to Signed Word format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvpsxws</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>843</td>
<td>VSX Vector Convert with round to zero Double-Precision to Signed Word format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvuxwsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>866</td>
<td>VSX Vector Convert with round Single-Precision to Signed Word format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvuxwsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>863</td>
<td>VSX Vector Convert with round Signed Word to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsxvuxsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>716</td>
<td>VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsxvuxsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>713</td>
<td>VSX Scalar Convert with round Signed Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsxvdpsxds</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>691</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsxvdpsxds</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>687</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsxvuxddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>715</td>
<td>VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsxvuxddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>712</td>
<td>VSX Scalar Convert with round Signed Doubleword to Double-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsxvpsxds</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>857</td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsxvpsxds</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>853</td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvuxxddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>865</td>
<td>VSX Vector Convert with round unsigned Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvuxxddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>862</td>
<td>VSX Vector Convert with round Signed Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvuxxddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>845</td>
<td>VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvuxxddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>841</td>
<td>VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvvuxddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>864</td>
<td>VSX Vector Convert with round unsigned Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvvuxddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>861</td>
<td>VSX Vector Convert with round Signed Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xrdpi</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>785</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward Zero XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xrdpiz</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>789</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xrdpip</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>788</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 24 of 30)
<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category 2</th>
<th>Always Optional Category 4</th>
<th>Mnemonic</th>
<th>Version 5</th>
<th>Privilege 6</th>
<th>Mode Dep 7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 1001 1000 0111</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xsrdivp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>797</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward -Infinity XX2-form</td>
</tr>
<tr>
<td>1000 1001 1000 0101</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xsabd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>656</td>
<td>VSX Scalar Absolute Double-Precision XX2-form</td>
</tr>
<tr>
<td>1000 1001 1000 0110</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xsnabdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>763</td>
<td>VSX Scalar Absolute Double-Precision XX2-form</td>
</tr>
<tr>
<td>1000 1001 1000 1001</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xsneg</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>764</td>
<td>VSX Scalar Negative Double-Precision XX2-form</td>
</tr>
<tr>
<td>1000 1001 1000 1010</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xvcvdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>840</td>
<td>VSX Scalar Convert with round Double-Precision to Single-Precision format XX2-form</td>
</tr>
<tr>
<td>1000 1001 1000 1011</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xvacvdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>851</td>
<td>VSX Scalar Convert Double-Precision to Single-Precision format XX2-form</td>
</tr>
<tr>
<td>1000 1001 1000 1100</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xvdvdvp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>820</td>
<td>VSX Scalar Convert Double-Precision to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>1000 1001 1000 1101</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xvnabsd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>921</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>1000 1001 1000 1110</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xvneg</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>922</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>1000 1001 1000 1111</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xvcvdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>820</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>1000 1001 1001 0000</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xvdvdvp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>921</td>
<td>VSX Scalar Convert Double-Precision to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>1000 1001 1001 0001</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xvtstd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>958</td>
<td>VSX Scalar Convert Double-Precision to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>1000 1001 1001 0010</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xvtsci</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>957</td>
<td>VSX Scalar Convert Double-Precision to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>1000 1001 1001 0011</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xsrsqrtfinp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>799</td>
<td>VSX Scalar Reciprocal Square Root Estimate Single-Precision XX2-form</td>
</tr>
<tr>
<td>1000 1001 1001 0100</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xrs</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>954</td>
<td>VSX Scalar Reciprocal Square Root Estimate Single-Precision XX2-form</td>
</tr>
<tr>
<td>1000 1001 1001 0101</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xrsr</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>978</td>
<td>VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form</td>
</tr>
<tr>
<td>1000 1001 1001 0110</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xrsrd</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>979</td>
<td>VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form</td>
</tr>
<tr>
<td>1000 1001 1001 1000</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>xstaq</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>740</td>
<td>VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 25 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 26 of 30)**
Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 27 of 30)
<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Book</th>
<th>Compliancy Subsets 2</th>
<th>Linux Optional Category 3</th>
<th>Always Optional Category 4</th>
<th>Mnemonic</th>
<th>Version 5</th>
<th>Privilege 6</th>
<th>Mode Dep 7</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFP dtstsfiq</td>
<td>V3.1</td>
<td>DFP</td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>216</td>
</tr>
<tr>
<td>xsadddp[e]</td>
<td>BFP128</td>
<td>xmuldp[c]</td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>666</td>
</tr>
<tr>
<td>xcmpgeqp</td>
<td>BFP128</td>
<td>xcmpgeqp</td>
<td>v3.1</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>759</td>
</tr>
<tr>
<td>xcmpgeqp</td>
<td>BFP128</td>
<td>xcmpgeqp</td>
<td>v3.1</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>671</td>
</tr>
<tr>
<td>xcmppgeqp</td>
<td>BFP128</td>
<td>xcmppgeqp</td>
<td>v3.1</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>682</td>
</tr>
<tr>
<td>xcmpgoqp</td>
<td>BFP128</td>
<td>xcmpgoqp</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>678</td>
</tr>
<tr>
<td>xcmppgoqp</td>
<td>BFP128</td>
<td>xcmppgoqp</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>678</td>
</tr>
<tr>
<td>xcmppgoqp</td>
<td>BFP128</td>
<td>xcmppgoqp</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>669</td>
</tr>
<tr>
<td>xsadddp[e]</td>
<td>BFP128</td>
<td>xmuldp[c]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>731</td>
</tr>
<tr>
<td>xsmsubdp[e]</td>
<td>BFP128</td>
<td>xsnmadddp</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>754</td>
</tr>
<tr>
<td>xsnmadddp</td>
<td>BFP128</td>
<td>xsnmadddp</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>773</td>
</tr>
<tr>
<td>xsnmadddp</td>
<td>BFP128</td>
<td>xsnmadddp</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>782</td>
</tr>
<tr>
<td>xsubdp[e]</td>
<td>BFP128</td>
<td>xsubdp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>809</td>
</tr>
<tr>
<td>xsdivdp[e]</td>
<td>BFP128</td>
<td>xsdivdp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>719</td>
</tr>
<tr>
<td>xsubdp[e]</td>
<td>BFP128</td>
<td>xsubdp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>661</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>738</td>
</tr>
<tr>
<td>xsubdp[e]</td>
<td>BFP128</td>
<td>xsubdp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>816</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>745</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>658</td>
</tr>
<tr>
<td>xsubdp[e]</td>
<td>BFP128</td>
<td>xsubdp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>818</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>763</td>
</tr>
<tr>
<td>xsubdp[e]</td>
<td>BFP128</td>
<td>xsubdp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>764</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>819</td>
</tr>
<tr>
<td>xsubdp[e]</td>
<td>BFP128</td>
<td>xsubdp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>803</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>705</td>
</tr>
<tr>
<td>xsubdp[e]</td>
<td>BFP128</td>
<td>xsubdp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>707</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>714</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>715</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>699</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>701</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>714</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>711</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>703</td>
</tr>
<tr>
<td>xadddp[e]</td>
<td>BFP128</td>
<td>xadddp[e]</td>
<td>v3.0</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>696</td>
</tr>
</tbody>
</table>

Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 28 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFP128 xscvdpqp v3.0</td>
<td>684</td>
<td>VSX Scalar Convert Double-Precision to Quad-Precision format X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFP128 xscvqpsdz v3.0</td>
<td>697</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFP128 xsiexpqp</td>
<td>724</td>
<td>VSX Scalar Insert Exponent Quad-Precision X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFP128 xsrqpi[x]</td>
<td>793</td>
<td>VSX Scalar Round to Quad-Precision Integer [with Inexact] Z23-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFP128 xsrqpxp</td>
<td>795</td>
<td>VSX Scalar Round Quad-Precision to Double-Extended Precision Z23-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtfsb1[.] P1</td>
<td>185</td>
<td>Move To FPSCR Bit 1 X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtfsb0[.] P1</td>
<td>185</td>
<td>Move To FPSCR Bit 0 X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtfsfi[.] P1</td>
<td>184</td>
<td>Move To FPSCR Field Immediate X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmrgow v2.07</td>
<td>162</td>
<td>Floating Merge Odd Word X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmrgew v2.07</td>
<td>162</td>
<td>Floating Merge Even Word X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mffs[.] P1</td>
<td>182</td>
<td>Move From FPSCR X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mffsce v3.0B</td>
<td>182</td>
<td>Move From FPSCR &amp; Clear Enables X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mffscdrn v3.0B</td>
<td>182</td>
<td>Move From FPSCR Control &amp; Set DRN X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mffscdrni v3.0B</td>
<td>183</td>
<td>Move From FPSCR Control &amp; Set DRN Immediate X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mffscrn v3.0B</td>
<td>183</td>
<td>Move From FPSCR Control &amp; Set RN X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mffscrni v3.0B</td>
<td>183</td>
<td>Move From FPSCR Control &amp; Set RN Immediate X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mffsl v3.0B</td>
<td>183</td>
<td>Move From FPSCR Lightweight X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtfsf[.] P1</td>
<td>184</td>
<td>Move To FPSCR Fields XFL-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fcpsgn[.] v2.05</td>
<td>161</td>
<td>Floating Copy Sign X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fneg[.] P1</td>
<td>161</td>
<td>Floating Negate X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmr[.] P1</td>
<td>161</td>
<td>Floating Move Register X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fnabs[.] P1</td>
<td>161</td>
<td>Floating Negative Absolute Value X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fabs[.] P1</td>
<td>161</td>
<td>Floating Absolute Value X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>frin[.] v2.02</td>
<td>178</td>
<td>Floating Round to Integer Nearest X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>friz[.] v2.02</td>
<td>178</td>
<td>Floating Round to Integer Toward Zero X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>frip[.] v2.02</td>
<td>178</td>
<td>Floating Round to Integer Plus X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>frim[.] v2.02</td>
<td>178</td>
<td>Floating Round to Integer Minus X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>frsp[.] P1</td>
<td>170</td>
<td>Floating Round to Single-Precision X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fctiw[.] P2</td>
<td>172</td>
<td>Floating Convert with round Double-Precision To Signed Word format X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fctiwu[.] v2.06</td>
<td>172</td>
<td>Floating Convert with round Double-Precision To Unsigned Word format X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fctid[.] PPC</td>
<td>170</td>
<td>Floating Convert with round Double-Precision To Signed Doubleword format X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fcfid[.] PPC</td>
<td>174</td>
<td>Floating Convert with round Signed Doubleword to Double-Precision format X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fctidu[.] v2.06</td>
<td>171</td>
<td>Floating Convert with round Double-Precision To Unsigned Doubleword format X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fctiwz[.] P2</td>
<td>173</td>
<td>Floating Convert with truncate Double-Precision To Signed Word format X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fctiwuz[.] v2.06</td>
<td>172</td>
<td>Floating Convert with truncate Double-Precision To Unsigned Word format X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fctidz[.] PPC</td>
<td>171</td>
<td>Floating Convert with truncate Double-Precision To Signed Doubleword format X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fctiduz[.] v2.06</td>
<td>172</td>
<td>Floating Convert with truncate Double-Precision To Unsigned Doubleword format X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fdiv[.] P1</td>
<td>164</td>
<td>Floating Divide A-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsub[.] P1</td>
<td>163</td>
<td>Floating Subtract A-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction1</td>
<td>Book</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction1</td>
<td>Compilany Subsets2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction1</td>
<td>Linux Optional Category3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction1</td>
<td>Always Optional Category4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Figure 91. Power ISA AS Instruction Set Sorted by Opcode (Sheet 30 of 30)

1. **Instruction**
   - Instruction bit that corresponds to a reserved field, must have a value of 0, otherwise invalid form.
   - Instruction bit that corresponds to an operand bit, may have a value of either 0 or 1.
   - Instruction bit that corresponds to an opcode bit having a value 0.
   - Instruction bit that corresponds to an opcode bit having a value 1.

2. **OpenPOWER Compliancy Subsets**
   - X... Instruction included in the Scalar Fixed-Point Compliancy subset.
   - .X.. Instruction included in the Scalar Fixed-Point + Floating-Point Compliancy subset.
   - ..X. Instruction included in the Linux Compliancy subset.
   - ...X Instruction included in the AIX Compliancy subset.

3. **Linux Optional Category**
   - AMO Instruction part of Atomic Memory Operations category.
   - BFP128 Instruction part of Quad-Precision Floating-Point category.
   - BHRB Instruction part of Branch History Rolling Buffer category.
   - DFP Instruction part of Decimal Floating-Point category.
   - EBB Instruction part of Event-Based Branch category.
   - MMA Instruction part of Matrix-Multiplication Assist category.

4. **Always Optional Category**
   - MMA Instruction part of Matrix-Multiplication Assist category.

5. **Version**
   - P1 Instruction introduced in POWER Architecture.
   - P2 Instruction introduced in POWER2 Architecture.
   - PPC Instruction introduced in PowerPC Architecture prior to v2.00.
   - v2.00 Instruction introduced in PowerPC Architecture Version 2.00.
   - v2.01 Instruction introduced in PowerPC Architecture Version 2.01.
   - v2.02 Instruction introduced in PowerPC Architecture Version 2.02.
   - v2.03 Instruction introduced in Power ISA Version 2.03.
   - v2.04 Instruction introduced in Power ISA Version 2.04.
   - v2.05 Instruction introduced in Power ISA Version 2.05.
   - v2.06 Instruction introduced in Power ISA Version 2.06.
   - v2.07 Instruction introduced in Power ISA Version 2.07.
   - v3.0 Instruction introduced in Power ISA Version 3.0.
   - v3.0B Instruction introduced in Power ISA Version 3.0B.
   - v3.0C Instruction introduced in Power ISA Version 3.0C.
   - v3.1 Instruction introduced in Power ISA Version 3.1.
### 6. Privilege

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Denotes an instruction that is treated as privileged.</td>
</tr>
<tr>
<td>O</td>
<td>Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor-privileged for mtspr), depending on the SPR or PMR number.</td>
</tr>
<tr>
<td>PI</td>
<td>Denotes an instruction that is illegal in privileged state.</td>
</tr>
<tr>
<td>HV</td>
<td>Denotes an instruction that can be executed only in hypervisor state.</td>
</tr>
<tr>
<td>UV</td>
<td>Denotes an instruction that can be executed only in ultravisor state.</td>
</tr>
</tbody>
</table>

### 7. Mode Dependency

Except as described below and in Section 1.10.3, “Effective Address Calculation”, in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

- **CT**: If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
- **SR**: The setting of status registers (such as XER and CR0) is mode-dependent.
- **SF=1**: The instruction can be executed only in 64-bit mode.
Appendix F. Power ISA Instruction Set Sorted by Version

This appendix lists all the instructions in the Power ISA, sorted in reverse order by ISA version, then by mnemonic.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>brd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>119</td>
<td>Byte-Reverse Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>brh</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>119</td>
<td>Byte-Reverse Halfword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>brw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>119</td>
<td>Byte-Reverse Word X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cfuged</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>106</td>
<td>Centrifuge Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cntzdcm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>105</td>
<td>Count Leading Zeros Doubleword under bit Mask X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cnttzdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>105</td>
<td>Count Trailing Zeros Doubleword under bit Mask X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dfp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dpp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>gnm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvkvq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>620</td>
<td>Load VSX Vector Special Value Quadword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvkp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>625</td>
<td>Load VSX Vector Pared DQ-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvrbr</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>626</td>
<td>Load VSX Vector Pared Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvrbrh</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>627</td>
<td>Load VSX Vector Rightmost Byte Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvrdx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>628</td>
<td>Load VSX Vector Rightmost Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvrvx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>629</td>
<td>Load VSX Vector Rightmost Halfword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mttbx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>630</td>
<td>Load VSX Vector Rightmost Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mttdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mttdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mttb</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtmsq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtmsq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtmsq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtmsq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>paddi</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>76</td>
<td>Prefix Add Immediate MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pdepd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>106</td>
<td>Parallel Bits Deposit Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pextd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>106</td>
<td>Parallel Bits Extract Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plbz</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>52</td>
<td>Prefix Load Byte and Zero MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pld</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>57</td>
<td>Prefix Load Doubleword MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pld</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>152</td>
<td>Prefix Load Floating-Point Double MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plfs</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>150</td>
<td>Prefix Load Floating-Point Single MLS:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plha</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>54</td>
<td>Prefix Load Halfword Algebraic MLS:D-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 1 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefixed Load Halfword and Zero MLS:D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>phhz</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>Prefixed Load Quadword 8LS:D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>Prefixed Load Word Algebraic 8LS:D-form</td>
<td></td>
<td></td>
<td></td>
<td>plwa</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>Prefixed Load Word and Zero MLS:D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plwz</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>Prefixed Load VSX Scalar Doubleword 8LS:D-form</td>
<td></td>
<td></td>
<td></td>
<td>pxsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Prefixed Load VSX Scalar Single-Precision 8LS:D-form</td>
<td></td>
<td></td>
<td></td>
<td>pxssp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>Prefixed Load VSX Vector 8LS:D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plxv</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Load VSX Vector Paired 8LS:D-form</td>
<td></td>
<td></td>
<td></td>
<td>plxvp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2nn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Positive accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Positive accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Positive accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Positive accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Positive accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Positive accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Positive accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Positive accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Negative accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Positive accumulate MMIRR:XX-form</td>
<td></td>
<td></td>
<td></td>
<td>pmxvbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Book</td>
<td>Compliancy Subsets</td>
<td>Linux Optional Category</td>
<td>Always Optional Category</td>
<td>Mnemonic</td>
<td>Version</td>
<td>Privilege</td>
<td>Mode Dep</td>
<td>Page</td>
<td>Name</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>--------------------</td>
<td>-------------------------</td>
<td>--------------------------</td>
<td>----------</td>
<td>---------</td>
<td>-----------</td>
<td>----------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvl64ger</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvl64germ</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvl64germp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvl64gerpn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxvl64gerpp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>879</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>891</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>891</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv16ger2s</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>893</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv16ger2ssp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>893</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv4ger8</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>883</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv4ger8pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>883</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv8ger4</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>886</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv8ger4pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>886</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv8ger4app</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>889</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv8ger4pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>889</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv8ger4app</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>889</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv8ger4app</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>889</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv8ger4app</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>889</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv8ger4app</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>889</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pmxv8ger4app</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>889</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pnop</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>130</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pstb</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>59</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pstid</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pstfd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>157</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pstfs</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>155</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>psth</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>60</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pstq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>66</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pstw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>61</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pstxs</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>638</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pstxspp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>642</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td></td>
<td>pstxv</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>644</td>
<td></td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 3 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>psbxvp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>654</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>selbc</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>129</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>selbcr</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>129</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>selbrc</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>129</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>selbrcr</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>129</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>spurn</td>
<td>v3.1</td>
<td></td>
<td>??</td>
<td>655</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>654</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvdx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>655</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvdrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>651</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvdrh</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>652</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvhrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>652</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctued</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>456</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctfrb</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>476</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctfrh</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>476</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctzdn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>449</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpequ</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>392</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpltq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>397</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmpltf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>397</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmplq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>401</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vcmplua</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>401</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctimbh</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>469</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctimbh</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>469</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vctzdn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>452</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvides</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>361</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvidesq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>363</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvidesw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>359</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdveuq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>361</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdveuq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>363</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdveuq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>359</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvides</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>360</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvidesq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>362</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvides</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>358</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvides</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>360</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvides</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>362</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdvides</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>358</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexpandbm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>466</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexpanddm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>467</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexpandh</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>466</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexpandq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>468</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexpandw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>467</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexddvx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>312</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexdvrnx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>312</td>
<td></td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 4 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliance Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxextdubvlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>309</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxextdubvrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>309</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxextduhvlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>310</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxextduhvrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>310</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxextduwvlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>311</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxextduwvrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>311</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxextractbm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>471</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxextractdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>471</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxextracthm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>473</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxextractqm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>473</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxextractwm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>472</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vexts2d2q</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>374</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vgnb</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>446</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinsblx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>315</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinsbrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>315</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinsblvix</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>320</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinsbrvix</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>320</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>319</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinsdtlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>318</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinsdx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>318</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinshlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>316</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinshrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>316</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinshlvix</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>321</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinshrvix</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>321</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinsw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>319</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinswlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>317</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinswrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>317</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinswvix</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>322</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinswvlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>322</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vinswvrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>322</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmodsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>365</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmodsq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>366</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmodsw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>364</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmodst</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>365</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmodstq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>366</td>
<td></td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 5 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmoduw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>364</td>
<td>Vector Modulo Unsigned Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmsumcud</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>357</td>
<td>Vector Multiply-Sum &amp; write Carry-out Unsigned Doubleword VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmulisd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>346</td>
<td>Vector Multiply Even Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmulisd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>345</td>
<td>Vector Multiply Even Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmultid</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>349</td>
<td>Vector Multiply High Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmultid</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>347</td>
<td>Vector Multiply High Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmultid</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>348</td>
<td>Vector Multiply High Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmultid</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>350</td>
<td>Vector Multiply Low Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmultid</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>346</td>
<td>Vector Multiply Odd Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmultid</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>345</td>
<td>Vector Multiply Odd Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vdepdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>454</td>
<td>Vector Parallel Bits Deposit Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpxextd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>455</td>
<td>Vector Parallel Bits Extract Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrlq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>405</td>
<td>Vector Rotate Left Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrlq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>412</td>
<td>Vector Rotate Left Quadword then Mask Insert VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrlq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>409</td>
<td>Vector Rotate Left Quadword then AND with Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vslbdi</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>298</td>
<td>Vector Shift Left Double by Bit Immediate VN-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsraq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>415</td>
<td>Vector Shift Right Algebraic Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vslbdi</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>421</td>
<td>Vector Shift Right Double by Bit Immediate VN-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>298</td>
<td>Vector Shift Right Double by Bit Immediate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>418</td>
<td>Vector Shift Right Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrhbr[]</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>474</td>
<td>Vector String Isolate Byte Left-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrhbr[]</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>475</td>
<td>Vector String Isolate Byte Right-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrhbr[]</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>475</td>
<td>Vector String Isolate Halfword Left-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrhbr[]</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>474</td>
<td>Vector String Isolate Halfword Right-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcmpnppq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>671</td>
<td>VSX Scalar Compare Equal Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcmpnppq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>673</td>
<td>VSX Scalar Compare Greater Than or Equal Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcmpnppq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>675</td>
<td>VSX Scalar Compare Greater Than Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsovgsqz</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>699</td>
<td>VSX Scalar Compare Convert with round to zero Quad-Precision Signed Quadword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsovgsqz</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>705</td>
<td>VSX Scalar Compare Convert with round to zero Quad-Precision to Unsigned Quadword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsovqppq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>711</td>
<td>VSX Scalar Compare Convert with round Signed Quadword to Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsovqppq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>715</td>
<td>VSX Scalar Compare Convert with round Unsigned Quadword to Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaxcnp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>738</td>
<td>VSX Scalar Maximum Type-C Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmincnp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>745</td>
<td>VSX Scalar Minimum Type-C Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>VSX Vector broadcast 16 GER (Rank-2 Update) XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>VSX Vector broadcast 16 GER (Rank-2 Update) Negative multiply, Negative accumulate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xbf16ger2pn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>VSX Vector broadcast 16 GER (Rank-2 Update) Negative multiply, Positive accumulate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xbf16ger2pn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>VSX Vector broadcast 16 GER (Rank-2 Update) Positive multiply, Negative accumulate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xbf16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>VSX Vector broadcast 16 GER (Rank-2 Update) Positive multiply, Positive accumulate XX3-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 6 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>31100 ... 11000 ... 11001 11102.</td>
<td>I ... X</td>
<td>MMA MMA</td>
<td>v3.1</td>
<td>xvcvbl8sp</td>
<td>v3.1</td>
<td>839</td>
<td>VSX Vector Convert bfloat16 to Single-Precision format XX2-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31100 ... 11000 ... 11001 11102.</td>
<td>I ... X</td>
<td>MMA MMA</td>
<td>v3.1</td>
<td>xvcvbl8</td>
<td>v3.1</td>
<td>850</td>
<td>VSX Vector Convert with round Single-Precision to bfloat16 format XX2-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11010 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf16ger2</td>
<td>v3.1</td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) XX2-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11010 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf16ger2en</td>
<td>v3.1</td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11010 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf16ger2np</td>
<td>v3.1</td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11010 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf16ger2pp</td>
<td>v3.1</td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11001 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf32ger</td>
<td>v3.1</td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) XXO-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11001 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf32germm</td>
<td>v3.1</td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX2-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11001 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf32germp</td>
<td>v3.1</td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11001 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf32gerpn</td>
<td>v3.1</td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11001 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf32gerpp</td>
<td>v3.1</td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11011 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf64ger</td>
<td>v3.1</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) XXO-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11011 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf64germm</td>
<td>v3.1</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX2-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11011 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf64germp</td>
<td>v3.1</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11011 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf64gerpn</td>
<td>v3.1</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11011 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf64gerpp</td>
<td>v3.1</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11011 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf16ger2</td>
<td>v3.1</td>
<td>911</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) XXO-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11011 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf16ger2en</td>
<td>v3.1</td>
<td>911</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) Negative multiply, Positive accumulate XX2-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11011 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf16ger2np</td>
<td>v3.1</td>
<td>911</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11011 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf16ger2pp</td>
<td>v3.1</td>
<td>911</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11100 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf4ger8</td>
<td>v3.1</td>
<td>883</td>
<td>VSX Vector 4-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate XX2-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11100 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf4ger8pp</td>
<td>v3.1</td>
<td>883</td>
<td>VSX Vector 4-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11100 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf6ger4</td>
<td>v3.1</td>
<td>886</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11100 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf6ger4pp</td>
<td>v3.1</td>
<td>886</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31101 ... 11001 ... 11100 11102.</td>
<td>I ... MMA</td>
<td>MMA</td>
<td>v3.1</td>
<td>xvf8ger4sp</td>
<td>v3.1</td>
<td>889</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturation Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31100 ... 11000 ... 11001 11102.</td>
<td>I ... X</td>
<td>MMA MMA</td>
<td>v3.1</td>
<td>xvtlsbb</td>
<td>v3.1</td>
<td>959</td>
<td>VSX Vector Test Least-Significant Bit by Byte XX2-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31100 ... 11000 ... 11001 11102.</td>
<td>I ... X</td>
<td>MMA MMA</td>
<td>v3.1</td>
<td>xbblendvb</td>
<td>v3.1</td>
<td>962</td>
<td>VSX Vector Blend Variable Byte 8RR/XX4-form</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 7 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxblendvd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>963</td>
<td>VSX Vector Blend Variable Doubleword 8RR:XX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxblendvh</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>962</td>
<td>VSX Vector Blend Variable Halfword 8RR:XX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxblendvw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>963</td>
<td>VSX Vector Blend Variable Word 8RR:XX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxeval</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>967</td>
<td>VSX Vector Evaluate 8RR-XX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xgpovdmb</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>970</td>
<td>VSX Vector Generate PCV from Byte Mask X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xgpovdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>976</td>
<td>VSX Vector Generate PCV from Doubleword Mask X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xgpovfm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>972</td>
<td>VSX Vector Generate PCV from Halfword Mask X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xgpovwm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>974</td>
<td>VSX Vector Generate PCV from Word Mask X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xmmfacc</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>983</td>
<td>VSX Move From Accumulator X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xmmfacc</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>984</td>
<td>VSX Move To Accumulator X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xmxmrm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>987</td>
<td>VSX Vector Permuted Extended 8RR,XX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xmxatcc</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>989</td>
<td>VSX Set Accumulator to Zero X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xssplit2dx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>992</td>
<td>VSX Vector Split Immediate32 Doubleword Indexed 8RR-D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xssplitdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>991</td>
<td>VSX Vector Split Immediate Double-Precision 8RR-D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsspliw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>992</td>
<td>VSX Vector Split Immediate Word 8RR,D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>magdru</td>
<td>v3.0C</td>
<td>UV</td>
<td></td>
<td>1328</td>
<td>Ultravisor Message Clear X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>magdru</td>
<td>v3.0C</td>
<td>UV</td>
<td></td>
<td>1327</td>
<td>Ultravisor Message SendX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>urldf</td>
<td>v3.0C</td>
<td>UV</td>
<td></td>
<td>1153</td>
<td>Ultravisor Return From Interrupt Doubleword XL-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addlex</td>
<td>v3.0B</td>
<td>80</td>
<td></td>
<td>80</td>
<td>Add Extended using alternate carry bit Z23-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfsscdm</td>
<td>v3.0B</td>
<td>162</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; Set DRN X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfsscdmi</td>
<td>v3.0B</td>
<td>183</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; Set DRN Immediate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfssce</td>
<td>v3.0B</td>
<td>182</td>
<td></td>
<td></td>
<td>Move From FPSCR &amp; Clear Enables X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfsscm</td>
<td>v3.0B</td>
<td>183</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; Set RN X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfsscmi</td>
<td>v3.0B</td>
<td>183</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; Set RN Immediate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfssf</td>
<td>v3.0B</td>
<td>183</td>
<td></td>
<td></td>
<td>Move From FPSCR Lightweight X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sibdiag</td>
<td>v.0B</td>
<td>P</td>
<td></td>
<td>1226</td>
<td>SLB Invalidate All Global X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmsumudm</td>
<td>v.0B</td>
<td>356</td>
<td></td>
<td></td>
<td>Vector Multiply-Sum Unsigned Doubleword Module VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addpcis</td>
<td>v.0</td>
<td>76</td>
<td></td>
<td></td>
<td>Add PC Immediate Shifted DX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdcdin</td>
<td>v.0</td>
<td>480</td>
<td></td>
<td></td>
<td>Decimal Convert From National VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdcfq</td>
<td>v.0</td>
<td>485</td>
<td></td>
<td></td>
<td>Decimal Convert From Signed Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdcf</td>
<td>v.0</td>
<td>481</td>
<td></td>
<td></td>
<td>Decimal Convert From Zoned VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdcppgn</td>
<td>v.0</td>
<td>489</td>
<td></td>
<td></td>
<td>Decimal Copy Sign VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdcsc</td>
<td>v.0</td>
<td>483</td>
<td></td>
<td></td>
<td>Decimal Convert To National VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdcsqiq</td>
<td>v.0</td>
<td>488</td>
<td></td>
<td></td>
<td>Decimal Convert To Signed Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdctz</td>
<td>v.0</td>
<td>484</td>
<td></td>
<td></td>
<td>Decimal Convert To Zoned VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcds</td>
<td>v.0</td>
<td>491</td>
<td></td>
<td></td>
<td>Decimal Shift VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdsetg</td>
<td>v.0</td>
<td>490</td>
<td></td>
<td></td>
<td>Decimal Set Sign VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdsr</td>
<td>v.0</td>
<td>493</td>
<td></td>
<td></td>
<td>Decimal Shift and Round VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdtrunc</td>
<td>v.0</td>
<td>494</td>
<td></td>
<td></td>
<td>Decimal Truncate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdus</td>
<td>v.0</td>
<td>492</td>
<td></td>
<td></td>
<td>Decimal Unsigned Shift VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdltrunc</td>
<td>v.0</td>
<td>495</td>
<td></td>
<td></td>
<td>Decimal Unsigned Truncate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmpregb</td>
<td>v.0</td>
<td>95</td>
<td></td>
<td></td>
<td>Compare Equal Byte X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmprb</td>
<td>v.0</td>
<td>94</td>
<td></td>
<td></td>
<td>Compare Ranged Byte X-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 8 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Name</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>............</td>
<td>.....</td>
<td>....................</td>
<td>..............</td>
<td>......</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>............</td>
<td>.....</td>
<td>....................</td>
<td>..............</td>
<td>......</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>............</td>
<td>.....</td>
<td>....................</td>
<td>..............</td>
<td>......</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>............</td>
<td>.....</td>
<td>....................</td>
<td>..............</td>
<td>......</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>............</td>
<td>.....</td>
<td>....................</td>
<td>..............</td>
<td>......</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>............</td>
<td>.....</td>
<td>....................</td>
<td>..............</td>
<td>......</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>............</td>
<td>.....</td>
<td>....................</td>
<td>..............</td>
<td>......</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>............</td>
<td>.....</td>
<td>....................</td>
<td>..............</td>
<td>......</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>............</td>
<td>.....</td>
<td>....................</td>
<td>..............</td>
<td>......</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>............</td>
<td>.....</td>
<td>....................</td>
<td>..............</td>
<td>......</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>............</td>
<td>.....</td>
<td>....................</td>
<td>..............</td>
<td>......</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
<td>........</td>
</tr>
</tbody>
</table>

**Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 9 of 30)**
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliance Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vabsdhhv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vabsdhh</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>378</td>
<td>v6.0</td>
</tr>
<tr>
<td>vabsdwv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vabsdw</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>379</td>
<td>v6.0</td>
</tr>
<tr>
<td>vpermmdv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vpermmd</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>461</td>
<td>v6.0</td>
</tr>
<tr>
<td>vclzlsbbv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vclzlsbb</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>453</td>
<td>v6.0</td>
</tr>
<tr>
<td>vcmpneb[v]v3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vcmpneb</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>398</td>
<td>v6.0</td>
</tr>
<tr>
<td>vcmpnew[v]v3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vcmpnew</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>400</td>
<td>v6.0</td>
</tr>
<tr>
<td>vcmpnewh[v]v3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vcmpnewh</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>399</td>
<td>v6.0</td>
</tr>
<tr>
<td>vctzbv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vctzb</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>450</td>
<td>v6.0</td>
</tr>
<tr>
<td>vctzdv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vctzd</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>451</td>
<td>v6.0</td>
</tr>
<tr>
<td>vcztzhv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vcztzh</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>450</td>
<td>v6.0</td>
</tr>
<tr>
<td>vctzbvb3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vctzsb</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>451</td>
<td>v6.0</td>
</tr>
<tr>
<td>vextsb2dv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vextsb2d</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>373</td>
<td>v6.0</td>
</tr>
<tr>
<td>vextsb2wv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vextsb2w</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>372</td>
<td>v6.0</td>
</tr>
<tr>
<td>vextsh2dv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vextsh2d</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>373</td>
<td>v6.0</td>
</tr>
<tr>
<td>vextsh2wv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vextsh2w</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>372</td>
<td>v6.0</td>
</tr>
<tr>
<td>vextuw2dv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vextuw2d</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>372</td>
<td>v6.0</td>
</tr>
<tr>
<td>vextublxv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vextublx</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>306</td>
<td>v6.0</td>
</tr>
<tr>
<td>vextubrxv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vextubrx</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>306</td>
<td>v6.0</td>
</tr>
<tr>
<td>vextuhnixv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vextuhnix</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>307</td>
<td>v6.0</td>
</tr>
<tr>
<td>vextuhxv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vextuhx</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>307</td>
<td>v6.0</td>
</tr>
<tr>
<td>vextuwixv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vextuwix</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>308</td>
<td>v6.0</td>
</tr>
<tr>
<td>vextuwrxv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vextuwrx</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>308</td>
<td>v6.0</td>
</tr>
<tr>
<td>vinsertbv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vinsertb</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>313</td>
<td>v6.0</td>
</tr>
<tr>
<td>vinsertdv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vinsertd</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>314</td>
<td>v6.0</td>
</tr>
<tr>
<td>vinserthv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vinserth</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>313</td>
<td>v6.0</td>
</tr>
<tr>
<td>vinserttwv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vinsertw</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>314</td>
<td>v6.0</td>
</tr>
<tr>
<td>vmul10cqv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vmul10cuq</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>487</td>
<td>v6.0</td>
</tr>
<tr>
<td>vmul10cquv3.0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>vmul10cuq</td>
<td>v3.0</td>
<td>x</td>
<td>x</td>
<td>488</td>
<td>v6.0</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 10 of 30)
<table>
<thead>
<tr>
<th>Instruction ¹</th>
<th>Book</th>
<th>Compliance/Subsets ²</th>
<th>Linux Optional Category ³</th>
<th>Always Optional Category ⁴</th>
<th>Mnemonic</th>
<th>Version  ⁵</th>
<th>Privilege ⁶</th>
<th>Mode Dep ⁷</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmul10uq</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>487</td>
<td>Vector Multiply-by-10 Unsigned Quadword VX-form</td>
</tr>
<tr>
<td>vnegd</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>371</td>
<td>Vector Negate Doubleword VX-form</td>
</tr>
<tr>
<td>vnegw</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>371</td>
<td>Vector Negate Word VX-form</td>
</tr>
<tr>
<td>vpermr</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>296</td>
<td>Vector Permute Right-indexed VX-form</td>
</tr>
<tr>
<td>vprtybd</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>459</td>
<td>Vector Parity Byte Doubleword VX-form</td>
</tr>
<tr>
<td>vprtybq</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>460</td>
<td>Vector Parity Byte Quadword VX-form</td>
</tr>
<tr>
<td>vprtybw</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>459</td>
<td>Vector Parity Byte Word VX-form</td>
</tr>
<tr>
<td>vrldmni</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>411</td>
<td>Vector Rotate Left Doubleword then Mask Insert VX-form</td>
</tr>
<tr>
<td>vrldrmni</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>408</td>
<td>Vector Rotate Left Doubleword then AND with Mask VX-form</td>
</tr>
<tr>
<td>vrlnrm</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>410</td>
<td>Vector Rotate Left Word then Mask Insert VX-form</td>
</tr>
<tr>
<td>vrlnrnm</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>407</td>
<td>Vector Rotate Left Word then AND with Mask VX-form</td>
</tr>
<tr>
<td>vslv</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>302</td>
<td>Vector Shift Left Variable VX-form</td>
</tr>
<tr>
<td>vsrv</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>302</td>
<td>Vector Shift Right Variable VX-form</td>
</tr>
<tr>
<td>xsabsqp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>658</td>
<td>VSX Scalar Absolute Quad-Precision X-form</td>
</tr>
<tr>
<td>xsaddsp[o]</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>666</td>
<td>VSX Scalar Add Quad-Precision [using round to Odd] X-form</td>
</tr>
<tr>
<td>xscmpeqdpd</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>670</td>
<td>VSX Scalar Compare Equal Double-Precision X32-form</td>
</tr>
<tr>
<td>xscmpeqdpqd</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>668</td>
<td>VSX Scalar Compare Exponents Double-Precision X32-form</td>
</tr>
<tr>
<td>xscmpeqdpqpd</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>669</td>
<td>VSX Scalar Compare Exponents Quad-Precision X32-form</td>
</tr>
<tr>
<td>xscmpeqdpdgp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>672</td>
<td>VSX Scalar Compare Greater Than or Equal Double-Precision X32-form</td>
</tr>
<tr>
<td>xscmpeqgdp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>674</td>
<td>VSX Scalar Compare Greater Than Double-Precision X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqpd</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>676</td>
<td>VSX Scalar Compare Ordered Quad-Precision X-form</td>
</tr>
<tr>
<td>xscmpeqgpqpdgp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>681</td>
<td>VSX Scalar Compare Unordered Quad-Precision X-form</td>
</tr>
<tr>
<td>xscmpeqgpdgpdgp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>682</td>
<td>VSX Scalar Compare Sign Quad-Precision X-form</td>
</tr>
<tr>
<td>xscmpeqgpdgpqpd</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>683</td>
<td>VSX Scalar Convert with round Double-Precision to Half-Precision format X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>684</td>
<td>VSX Scalar Convert Double-Precision to Quad-Precision format X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqpd</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>695</td>
<td>VSX Scalar Convert Half-Precision to Double-Precision format X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqpdgp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>696</td>
<td>VSX Scalar Convert with round Double-Precision to Double-Precision format [using round to Odd] X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqdpqpdgp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>697</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqdpqdpqdp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>701</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqdpqdpqdp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>703</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqdpqdpqdp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>707</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqdpqdpqdp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>714</td>
<td>VSX Scalar Convert Signed Doubleword to Quad-Precision format X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqdpqdpqdp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>714</td>
<td>VSX Scalar Convert Signed Doubleword to Quad-Precision format X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqdpqdpqdp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>719</td>
<td>VSX Scalar Divide Quad-Precision [using round to Odd] X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqdpqdpqdp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>723</td>
<td>VSX Scalar Insert Exponent Double-Precision X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqdpqdpqdp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>724</td>
<td>VSX Scalar Insert Exponent Quad-Precision X32-form</td>
</tr>
<tr>
<td>xscmpeqgpqdpqdpqdpqdp</td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>731</td>
<td>VSX Scalar Multiply-Add Quad-Precision [using round to Odd] X32-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 11 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaxcdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Maximum Type-C Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaxdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Maximum Type-J Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmincdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Minimum Type-C Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsminjp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Minimum Type-J Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmuljdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Multiply-J Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnegjdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-J Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmaddjdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negative Multiply-Add-J Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmsubjdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negative Multiply-Subtract-J Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnnpq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[1]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[2]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[3]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[4]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[5]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[6]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[7]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[8]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[9]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[10]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[12]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[13]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[14]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[15]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[16]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[17]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[18]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[19]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[20]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[21]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[22]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[23]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[24]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[25]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[26]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[27]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[28]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[29]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmpq[30]v3.0</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Negate-Multiply-Quad-P Double-Precision XX3-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 12 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets 1</th>
<th>Linux Optional Category 2</th>
<th>Always Optional Category 3</th>
<th>Mnemonic</th>
<th>Version 4</th>
<th>Privilege 5</th>
<th>Mode Dep 6</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100......</td>
<td>000010</td>
<td>I</td>
<td></td>
<td></td>
<td>xxinseuw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>969</td>
<td>VSX Vector Insert Word VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>xperm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>985</td>
<td>VSX Vector Permute XX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>001110</td>
<td>I</td>
<td></td>
<td></td>
<td>xpermrr</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>985</td>
<td>VSX Vector Permute Right-indexed XX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>001110</td>
<td>G</td>
<td></td>
<td></td>
<td>xsspx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>991</td>
<td>VSX Vector Split Immediate Byte X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000001</td>
<td>I</td>
<td></td>
<td></td>
<td>bcdadd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>478</td>
<td>Decimal Add Modulo VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000001</td>
<td>I</td>
<td></td>
<td></td>
<td>bcdsubb</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>478</td>
<td>Decimal Subtract Modulo VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000001</td>
<td>I</td>
<td></td>
<td></td>
<td>bctar[t]</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>43</td>
<td>Branch Conditional to Branch Target Address Register X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>BHRB</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Clear BHRB X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>finrgew</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>162</td>
<td>Floating Merge Even Word X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>finrgow</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>162</td>
<td>Floating Merge Odd Word X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000001</td>
<td>I</td>
<td></td>
<td></td>
<td>icbt</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>1052</td>
<td>Instruction Cache Block Touch X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000001</td>
<td>I</td>
<td></td>
<td></td>
<td>lgax</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Load Quadword And Reserve Indexed X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000001</td>
<td>I</td>
<td></td>
<td></td>
<td>lxiwax</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>613</td>
<td>Load VSX Scalar as Integer Word Algebraic Indexed X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000001</td>
<td>I</td>
<td></td>
<td></td>
<td>lxiwzx</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>614</td>
<td>Load VSX Scalar as Integer Word &amp; Zero Indexed X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000001</td>
<td>I</td>
<td></td>
<td></td>
<td>lxsspx</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>616</td>
<td>Load VSX Scalar Single-Precision Indexed X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>BHRB</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Move From BHRB VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>mitevrd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>120</td>
<td>Move From VSR Doubledword X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>mitevrez</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>121</td>
<td>Move From VSR Word and Zero X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>msgalnum</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>1329</td>
<td>Message Clear X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>msgand</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>1328</td>
<td>Message Send X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>msgland</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>1330</td>
<td>Message Send Privileged X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>mtvrd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>121</td>
<td>Move To VSR Doubleword X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>mtvrsna</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>122</td>
<td>Move To VSR Word Algebraic X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>mtvrsnz</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>122</td>
<td>Move To VSR Word and Zero X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>EBB</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Return from Event Based Branch X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>stqck</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Store Quadword Conditional Indexed X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>stbsiwx</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>641</td>
<td>Store VSX Scalar as Integer Word Indexed X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>stbsiwx</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>643</td>
<td>Store VSX Scalar Single-Precision Indexed X-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vaddsvx</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>330</td>
<td>Vector Add &amp; write Carry Unsized Quadword VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vaddecuq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>330</td>
<td>Vector Add Extended &amp; write Carry Unsized Quadword VA-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vaddsvq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>329</td>
<td>Vector Add Extended Unsized Quadword Modulo VA-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vaddsvq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>326</td>
<td>Vector Add Unsized Doubleword Modulo VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vaddsvq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>329</td>
<td>Vector Add Unsized Quadword Modulo VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vופinheritdoc</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>462</td>
<td>Vector Bit Permuted Quadword VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vופphetamine</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>435</td>
<td>Vector AES Cipher Last VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vופ.hamcrest</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>435</td>
<td>Vector AES Cipher Last VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vופ liegt</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>447</td>
<td>Vector Count Leading Zeros Byte VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vופ liegt</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>449</td>
<td>Vector Count Leading Zeros Doubleword VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vופ liegt</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>447</td>
<td>Vector Count Leading Zeros Halfword VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vופ liegt</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>448</td>
<td>Vector Count Leading Zeros Word VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vופ 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>391</td>
<td>Vector Compare Equal Unsized Doubleword VC-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>v[op 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>396</td>
<td>Vector Compare Greater Than Signed Doubleword VC-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>v[op 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁 궁</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>396</td>
<td>Vector Compare Greater Than Unsized Doubleword VC-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vevev</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical Equivalence VX-form</td>
</tr>
<tr>
<td>000100......</td>
<td>000110</td>
<td>I</td>
<td></td>
<td></td>
<td>vgbbd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>445</td>
<td>Vector Gather Bits by Bytes by Doubleword VX-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 13 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0010 0011</td>
<td>0010</td>
<td>...</td>
<td>vmaxsd</td>
<td>v.2.07</td>
<td>383</td>
<td></td>
<td></td>
<td></td>
<td>Vector Maximum Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0010 0011</td>
<td>0010</td>
<td>...</td>
<td>vmaxud</td>
<td>v.2.07</td>
<td>383</td>
<td></td>
<td></td>
<td></td>
<td>Vector Maximum Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0010 0011</td>
<td>0010</td>
<td>...</td>
<td>vminsnd</td>
<td>v.2.07</td>
<td>387</td>
<td></td>
<td></td>
<td></td>
<td>Vector Minimum Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0010 0011</td>
<td>0010</td>
<td>...</td>
<td>vminud</td>
<td>v.2.07</td>
<td>387</td>
<td></td>
<td></td>
<td></td>
<td>Vector Minimum Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vmmovw</td>
<td>v.2.07</td>
<td>292</td>
<td></td>
<td></td>
<td></td>
<td>Vector Merge Even Word VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vmmovw</td>
<td>v.2.07</td>
<td>292</td>
<td></td>
<td></td>
<td></td>
<td>Vector Merge Odd Word VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vmultlsw</td>
<td>v.2.07</td>
<td>343</td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply Even Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vmultlsw</td>
<td>v.2.07</td>
<td>343</td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply Even Unsigned Word VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vmultlsw</td>
<td>v.2.07</td>
<td>343</td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply Odd Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vmultlsw</td>
<td>v.2.07</td>
<td>344</td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply Odd Unsigned Word VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vmultw</td>
<td>v.2.07</td>
<td>347</td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply Unsigned Word Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vnan</td>
<td>v.2.07</td>
<td>403</td>
<td></td>
<td></td>
<td></td>
<td>Vector Logical NAND VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vncipher</td>
<td>v.2.07</td>
<td>438</td>
<td></td>
<td></td>
<td></td>
<td>Vector AES Inverse Cipher VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vncipherlast</td>
<td>v.2.07</td>
<td>438</td>
<td></td>
<td></td>
<td></td>
<td>Vector AES Inverse Cipher Last VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vncrc</td>
<td>v.2.07</td>
<td>403</td>
<td></td>
<td></td>
<td></td>
<td>Vector Logical OR with Complement VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0110</td>
<td>...</td>
<td>vpermxor</td>
<td>v.2.07</td>
<td>444</td>
<td></td>
<td></td>
<td></td>
<td>Vector Permute &amp; Exclusive-OR VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0111</td>
<td>...</td>
<td>vpksds</td>
<td>v.2.07</td>
<td>281</td>
<td></td>
<td></td>
<td></td>
<td>Vector Pack Signed Doubleword Signed Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0111</td>
<td>...</td>
<td>vpksds</td>
<td>v.2.07</td>
<td>281</td>
<td></td>
<td></td>
<td></td>
<td>Vector Pack Signed Doubleword Unsigned Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0111</td>
<td>...</td>
<td>vpkudum</td>
<td>v.2.07</td>
<td>264</td>
<td></td>
<td></td>
<td></td>
<td>Vector Pack Unsigned Doubleword Unsigned Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0111</td>
<td>...</td>
<td>vpkudus</td>
<td>v.2.07</td>
<td>264</td>
<td></td>
<td></td>
<td></td>
<td>Vector Pack Unsigned Doubleword Unsigned Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0100</td>
<td>...</td>
<td>vpmnsumb</td>
<td>v.2.07</td>
<td>440</td>
<td></td>
<td></td>
<td></td>
<td>Vector Polynomial Multiply-Sum Byte VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0100</td>
<td>...</td>
<td>vpmnsumd</td>
<td>v.2.07</td>
<td>443</td>
<td></td>
<td></td>
<td></td>
<td>Vector Polynomial Multiply-Sum Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0100</td>
<td>...</td>
<td>vpmnsumh</td>
<td>v.2.07</td>
<td>441</td>
<td></td>
<td></td>
<td></td>
<td>Vector Polynomial Multiply-Sum Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0100</td>
<td>...</td>
<td>vpmnsumw</td>
<td>v.2.07</td>
<td>442</td>
<td></td>
<td></td>
<td></td>
<td>Vector Polynomial Multiply-Sum Word VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0001</td>
<td>...</td>
<td>vppopcntb</td>
<td>v.2.07</td>
<td>457</td>
<td></td>
<td></td>
<td></td>
<td>Vector Population Count Byte VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0001</td>
<td>...</td>
<td>vppopcnt</td>
<td>v.2.07</td>
<td>458</td>
<td></td>
<td></td>
<td></td>
<td>Vector Population Count Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0001</td>
<td>...</td>
<td>vppopcnth</td>
<td>v.2.07</td>
<td>457</td>
<td></td>
<td></td>
<td></td>
<td>Vector Population Count Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0001</td>
<td>...</td>
<td>vppopcntw</td>
<td>v.2.07</td>
<td>458</td>
<td></td>
<td></td>
<td></td>
<td>Vector Population Count Word VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vrlid</td>
<td>v.2.07</td>
<td>405</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsacl</td>
<td>v.2.07</td>
<td>437</td>
<td></td>
<td></td>
<td></td>
<td>Vector AES SubBytes VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0011</td>
<td>...</td>
<td>vhasigmad</td>
<td>v.2.07</td>
<td>438</td>
<td></td>
<td></td>
<td></td>
<td>Vector SHA-512 Sigma Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0011</td>
<td>...</td>
<td>vhasigmadw</td>
<td>v.2.07</td>
<td>439</td>
<td></td>
<td></td>
<td></td>
<td>Vector SHA-512 Sigma Word VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0010</td>
<td>...</td>
<td>vadd</td>
<td>v.2.07</td>
<td>414</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Left Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0010</td>
<td>...</td>
<td>vsal</td>
<td>v.2.07</td>
<td>420</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Algebraic Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0010</td>
<td>...</td>
<td>vssrad</td>
<td>v.2.07</td>
<td>417</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsbsal</td>
<td>v.2.07</td>
<td>338</td>
<td></td>
<td></td>
<td></td>
<td>Vector Subtract &amp; write Carry-out Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsbsal</td>
<td>v.2.07</td>
<td>338</td>
<td></td>
<td></td>
<td></td>
<td>Vector Subtract &amp; write Carry-out Unaligned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsbsal</td>
<td>v.2.07</td>
<td>337</td>
<td></td>
<td></td>
<td></td>
<td>Vector Subtract Extended &amp; write Carry-out Unaligned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsbsal</td>
<td>v.2.07</td>
<td>337</td>
<td></td>
<td></td>
<td></td>
<td>Vector Subtract Extended Unaligned Doubleword Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsbsal</td>
<td>v.2.07</td>
<td>334</td>
<td></td>
<td></td>
<td></td>
<td>Vector Subtract Unaligned Doubleword Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsbsal</td>
<td>v.2.07</td>
<td>337</td>
<td></td>
<td></td>
<td></td>
<td>Vector Subtract Unaligned Doubleword Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsbsal</td>
<td>v.2.07</td>
<td>287</td>
<td></td>
<td></td>
<td></td>
<td>Vector Unpack High Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsbsal</td>
<td>v.2.07</td>
<td>287</td>
<td></td>
<td></td>
<td></td>
<td>Vector Unpack Low Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsbsal</td>
<td>v.2.07</td>
<td>664</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Add Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsbsal</td>
<td>v.2.07</td>
<td>666</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form</td>
</tr>
<tr>
<td></td>
<td>0111 0011</td>
<td>0000</td>
<td>...</td>
<td>vsbsal</td>
<td>v.2.07</td>
<td>710</td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Single-Precision to Double-Precision format Non-signalling XX2-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 14 of 30)
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Book</th>
<th>Compliant Subsets2</th>
<th>Linux Optional Category3</th>
<th>Always Optional Category4</th>
<th>Mnemonic</th>
<th>Version5</th>
<th>Privilege6</th>
<th>Mode Dep7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xscvxdsnsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>713</td>
<td>VSX Scalar Convert round Signed Doubleword to Single-Precision format X2X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xscvuxdsnsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>716</td>
<td>VSX Scalar Convert round Signed Doubleword to Single-Precision format X2X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xddivsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>721</td>
<td>VSX Scalar Divide Single-Precision X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaddsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>728</td>
<td>VSX Scalar Multiply-Add Type-A Single-Precision X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaddmsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>728</td>
<td>VSX Scalar Multiply-Add Type-M Single-Precision X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmsubasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>751</td>
<td>VSX Scalar Multiply-Subtract Type-A Single-Precision X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmsubmsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>779</td>
<td>VSX Scalar Multiply-Subtract Type-M Single-Precision X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsresp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>791</td>
<td>VSX Scalar Reciprocal Estimate Single-Precision X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsresp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>797</td>
<td>VSX Scalar Reciprocal Estimate Single-Precision X2X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsresgp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>799</td>
<td>VSX Scalar Reciprocal Square Root Estimate Single-Precision X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xssqrtsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>805</td>
<td>VSX Scalar Square Root Single-Precision X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xssubsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>811</td>
<td>VSX Scalar Subtract Single-Precision X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxleqv</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>979</td>
<td>VSX Vector Logical Equivalence X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxleqv</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>979</td>
<td>VSX Vector Logical NAND X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxlorc</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>980</td>
<td>VSX Vector Logical OR with Complement X0X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addg6s</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>118</td>
<td>Add and Generate Sixes X0-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bpermnd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>105</td>
<td>Bit-Permute Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cbcbldt</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>117</td>
<td>Convert Binary Coded Decimal To Declets X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cbcbcdt</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>117</td>
<td>Convert Declets to Binary Coded Decimal X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcflix[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>231</td>
<td>DPF Convert From Fixed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dived()</td>
<td>v2.06</td>
<td>SR 90</td>
<td></td>
<td></td>
<td>Divide Doubleword Extended X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dived()</td>
<td>v2.06</td>
<td>SR 90</td>
<td></td>
<td></td>
<td>Divide Doubleword Extended &amp; record OV X0-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dived()</td>
<td>v2.06</td>
<td>SR 90</td>
<td></td>
<td></td>
<td>Divide Doubleword Extended Signed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dived()</td>
<td>v2.06</td>
<td>SR 90</td>
<td></td>
<td></td>
<td>Divide Doubleword Extended &amp; record OV X0-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dived()</td>
<td>v2.06</td>
<td>SR 83</td>
<td></td>
<td></td>
<td>Divide Word Extended X0-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dived()</td>
<td>v2.06</td>
<td>SR 83</td>
<td></td>
<td></td>
<td>Divide Word Extended &amp; record OV X0-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dived()</td>
<td>v2.06</td>
<td>SR 83</td>
<td></td>
<td></td>
<td>Divide Word Extended Signed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fclfid[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>175</td>
<td>Floating Convert with round Signed Doubleword to Single-Precision format X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fclfid[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>175</td>
<td>Floating Convert with round Signed Doubleword to Double-Precision format X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fclfid[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>176</td>
<td>Floating Convert with round Signed Doubleword to Single-Precision format X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fclidu[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>171</td>
<td>Floating Convert with round Double-Precision To Unsigned Doubleword format X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fcliduz[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>172</td>
<td>Floating Convert with Truncate Double-Precision To Unsigned Doubleword format X-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 15 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compilatory Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fctiw2[,]</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>173</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fctiw2[,]</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>172</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fsdav</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>168</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fsgrt</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>167</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lbax</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>1077</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lbax</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>69</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fliwvx</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>153</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>iahx</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>1078</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxaxd</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>611</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvd2x</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>619</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvddsn</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>633</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxw4x</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>635</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lpcntnld</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>104</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lpcnltw</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>103</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbcx</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>1079</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdbnx</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>69</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stbdcx</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>1080</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stbdcx</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>639</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stbdvdx</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>646</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stxw4x</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>653</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xabsdpl</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>658</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xadddp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>659</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcmpnpdp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>676</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcmpnpdp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>679</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcpيضغدپ</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>682</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvdpsdp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>685</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvdpsds</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>687</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvdpsxws</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>689</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvdpxuws</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>691</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvdpxuxs</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>693</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvvadp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>709</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvxvddcp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>712</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvuxdcp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>715</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xdivvdcp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>717</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaddadp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>725</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmamdmdp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>725</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmadmp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>734</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmindp</td>
<td>v.2.06</td>
<td></td>
<td></td>
<td>741</td>
<td></td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 16 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliance Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110</td>
<td>1...</td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xmsubadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>748</td>
<td>VSX Scalar Multiply-Subtract Type-A Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td>1...</td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xmsubmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>748</td>
<td>VSX Scalar Multiply-Subtract Type-M Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xsmuldp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>757</td>
<td>VSX Scalar Multiply Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xsnabsdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>763</td>
<td>VSX Scalar Negative Absolute Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xsnegdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>764</td>
<td>VSX Scalar Negative Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td>1...</td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xsnaddadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>765</td>
<td>VSX Scalar Negative Multiply-Add Type-A Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td>1...</td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xsnaddmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>765</td>
<td>VSX Scalar Negative Multiply-Add Type-M Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xsnmsubadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>776</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xsnmsubmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>776</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xsrddpi</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>785</td>
<td>VSX Scalar Round to Double-Precision Integer using round to Nearest Away X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xsrddpm</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>786</td>
<td>VSX Scalar Round to Double-Precision Integer exact using Current rounding mode X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xsrddpip</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>787</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward -Infinity X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xsrddpiz</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>788</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward Infinity X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xsredp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>789</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward Nonzero X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xsrsqrtel</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>790</td>
<td>VSX Scalar Reciprocal Estimate Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xsrsqrtel</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>791</td>
<td>VSX Scalar Reciprocal Square Root Estimate Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td>1...</td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xssqrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>801</td>
<td>VSX Scalar Square Root Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td>1...</td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xssubdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>807</td>
<td>VSX Scalar Subtract Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xstdivdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>813</td>
<td>VSX Scalar Test for software Divide Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xstsqrtel</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>814</td>
<td>VSX Scalar Test for software Square Root Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xsvabsdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>820</td>
<td>VSX Vector Absolute Value Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xsvabsdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>820</td>
<td>VSX Vector Absolute Value Single-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xsvaddsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>821</td>
<td>VSX Vector Add Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xsvaddsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>825</td>
<td>VSX Vector Add Single-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xvcmpnegdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>832</td>
<td>VSX Vector Compare Equal To Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xvcmpeqsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>833</td>
<td>VSX Vector Compare Equal To Single-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xvcmpeqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>834</td>
<td>VSX Vector Compare Greater Than or Equal To Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xvcmpeqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>835</td>
<td>VSX Vector Compare Greater Than or Equal To Single-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xvcmpeqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>836</td>
<td>VSX Vector Compare Greater Than Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xvcmpeqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>837</td>
<td>VSX Vector Compare Greater Than Single-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xvcpagndp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>838</td>
<td>VSX Vector Copy Sign Double-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0111 011.</td>
<td></td>
<td></td>
<td>xvcpgndsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>838</td>
<td>VSX Vector Copy Sign Single-Precision X2Q-form</td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>0110 011.</td>
<td></td>
<td></td>
<td>xvcvdpisp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>840</td>
<td>VSX Vector Convert with round to Nearest Double-Precision to Single-Precision format X2Q-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 17 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvcvdpdsxds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>841</td>
<td>VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format XX2-form</td>
</tr>
<tr>
<td>xvcvdpdsxws</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>843</td>
<td>VSX Vector Convert with round to zero Double-Precision to Signed Word format XX2-form</td>
</tr>
<tr>
<td>xvcvdpuxds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>845</td>
<td>VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form</td>
</tr>
<tr>
<td>xvcvdpuxws</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>847</td>
<td>VSX Vector Convert with round to zero Double-Precision to Unsigned Word format XX2-form</td>
</tr>
<tr>
<td>xvcvpspxs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>851</td>
<td>VSX Vector Convert Single-Precision to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>xvcvpspxw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>853</td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format XX2-form</td>
</tr>
<tr>
<td>xvcvpspxdx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>855</td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Word format XX2-form</td>
</tr>
<tr>
<td>xvcvpspxdwx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>857</td>
<td>VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format XX2-form</td>
</tr>
<tr>
<td>xvcvpspxdxs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>859</td>
<td>VSX Vector Convert with round to zero Single-Precision to Unsigned Word format XX2-form</td>
</tr>
<tr>
<td>xcvovspdxdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>861</td>
<td>VSX Vector Convert with round Signed Doubleword to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>xcvovspdxsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>862</td>
<td>VSX Vector Convert with round Signed Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td>xcvovspdxsdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>863</td>
<td>VSX Vector Convert Signed Word to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>xcvovspdxsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>864</td>
<td>VSX Vector Convert with round Signed Word to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>xcvovspdxs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>865</td>
<td>VSX Vector Convert with round Signed Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td>xcvovspdxspp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>866</td>
<td>VSX Vector Convert Unsigned Word to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>xcvovspdxsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>867</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td>xcvovspdxsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>868</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td>xcvovspdxs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>869</td>
<td>VSX Vector Divide Double-Precision format XX3-form</td>
</tr>
<tr>
<td>xcvovspdxspp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>870</td>
<td>VSX Vector Divide Single-Precision format XX3-form</td>
</tr>
<tr>
<td>xvmaddadp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>897</td>
<td>VSX Vector Multiply-Add type A Double-Precision format XX3-form</td>
</tr>
<tr>
<td>xvmaddasp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>900</td>
<td>VSX Vector Multiply-Add type A Single-Precision format XX3-form</td>
</tr>
<tr>
<td>xvmaddmdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>903</td>
<td>VSX Vector Maximum Double-Precision format XX3-form</td>
</tr>
<tr>
<td>xvmaddmsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>905</td>
<td>VSX Vector Minimum Double-Precision format XX3-form</td>
</tr>
<tr>
<td>xvmsubadp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>907</td>
<td>VSX Vector Minimum Single-Precision format XX3-form</td>
</tr>
<tr>
<td>xvmsubasp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>909</td>
<td>VSX Vector Multiply-Subtract Type A Double-Precision format XX3-form</td>
</tr>
<tr>
<td>xvmsubmdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>911</td>
<td>VSX Vector Multiply-Subtract Type A Single-Precision format XX3-form</td>
</tr>
<tr>
<td>xvmsubmsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>914</td>
<td>VSX Vector Multiply-Subtract Type M Double-Precision format XX3-form</td>
</tr>
<tr>
<td>xvmsubmsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>911</td>
<td>VSX Vector Multiply-Subtract Type M Single-Precision format XX3-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 18 of 30)
<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Book</th>
<th>Compliance Subsets 2</th>
<th>Linux Optional Category 3</th>
<th>Always Optional Category 4</th>
<th>Mnemonic</th>
<th>Version 5</th>
<th>Privilege 6</th>
<th>Mode Dep 7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvmsubmsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply-Subtract Type-M Single-Precision X3-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvmulp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply Double-Precision XX3-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvnabsdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply Single-Precision XX3-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvnabspsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply Absolute Double-Precision X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvnabsss</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply Absolute Single-Precision X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvnegdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply Double-Precision XX2-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvnegsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply Single-Precision X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvmaddadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply-Add Type-A Double-Precision X03-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvmaddasp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply-Add Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvmaddndp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply-Add Type-M Double-Precision X03-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvmaddmap</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply-Add Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvmmsubdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply-Subtract Type-A Double-Precision X03-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvmmsubasp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply-Subtract Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvmmsubmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply-Subtract Type-M Double-Precision X03-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvmmsubmsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Multiply-Subtract Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvdrdpi</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Round to Double-Precision Integer using round to Nearest Away XX2-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvdrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Round to Double-Precision Integer using Current rounding mode X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvdrdpm</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Round to Double-Precision Integer using round toward Infinity X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvdrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Round to Double-Precision Integer using round toward Infinity X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvdrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Round to Double-Precision Integer using round toward Infinity X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvdrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Round to Double-Precision Integer using round toward Infinity X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvdrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Round to Double-Precision Integer using round toward Infinity X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvdrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Round to Double-Precision Integer using round toward Infinity X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvdrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Round to Double-Precision Integer using round toward Infinity X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvdrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Round to Double-Precision Integer using round toward Infinity X02-form</td>
</tr>
<tr>
<td>31110</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>I ...</td>
<td>xvdrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VSX Vector Round to Double-Precision Integer using round toward Infinity X02-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 19 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxvsubsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>952</td>
<td>VSX Vector Subtract Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvtdvdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>954</td>
<td>VSX Vector Test for software Divide Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvtdvvp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>955</td>
<td>VSX Vector Test for software Divide Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvtqrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>956</td>
<td>VSX Vector Test for software Square Root Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvtqrstp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>956</td>
<td>VSX Vector Test for software Square Root Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxland</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>978</td>
<td>VSX Vector Logical AND XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxlandc</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>978</td>
<td>VSX Vector Logical AND with Complement XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnor</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>980</td>
<td>VSX Vector Logical NOK XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxor</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>981</td>
<td>VSX Vector Logical OR XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxxor</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>981</td>
<td>VSX Vector Logical XOR XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnorhw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>982</td>
<td>VSX Vector Merge High Word XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxormlw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>982</td>
<td>VSX Vector Merge Low Word XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxpermld</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>988</td>
<td>VSX Vector Permuted Doubleword Immediate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsel</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>988</td>
<td>VSX Vector Select XX4-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsidwi</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>990</td>
<td>VSX Vector Shift Left Double by Word Immediate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxspithw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>993</td>
<td>VSX Vector Spat Word XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmpb</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>102</td>
<td>DFP Compare Bytes X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>daddds</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>206</td>
<td>DFP Add X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dadddq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>206</td>
<td>DFP Add Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dclfxxx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>231</td>
<td>DFP Convert From Fixed Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcmipo</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>272</td>
<td>DFP Compare Ordered X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcmipoq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>272</td>
<td>DFP Compare Ordered Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcmupa</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>271</td>
<td>DFP Compare Unordered X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcmpq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>271</td>
<td>DFP Compare Unordered Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcmpxq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>229</td>
<td>DFP Convert To DFP Long X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcmpxuq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcmxuq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcmoxq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To DFP Extended X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcmpoxq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>229</td>
<td>DFP Convert To DFP Extended Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ddedpdq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>235</td>
<td>DFP Decode DPD To BCD X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ddedpdq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>235</td>
<td>DFP Decode DPD To BCD Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ddvqj</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>209</td>
<td>DFP Divide X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ddvql</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>209</td>
<td>DFP Divide Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dencboj</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>235</td>
<td>DFP Encode BCD To DPD X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dencboq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>235</td>
<td>DFP Encode BCD To DPD Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>diexo</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Insert Biased Exponent X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dixj</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Insert Biased Exponent Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmulj</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>208</td>
<td>DFP Multiply X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmulpj</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>208</td>
<td>DFP Multiply Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmpaj</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>219</td>
<td>DFP Quantize Z23-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmpaq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>217</td>
<td>DFP Quantize Immediate Z23-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmpaqj</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>217</td>
<td>DFP Quantize Immediate Z23-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmpaqx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>219</td>
<td>DFP Quantize Quad Z23-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmpaqxx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>219</td>
<td>DFP Quantize Quad Z23-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmpqj</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>230</td>
<td>DFP Round To DFP Long X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmpqrt</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>226</td>
<td>DFP Round To FP Integer Without Inexact Z23-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmpqj</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>226</td>
<td>DFP Round To FP Integer Without Inexact Quad Z23-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmpqix</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>224</td>
<td>DFP Round To FP Integer With Inexact Z23-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmpqix</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>224</td>
<td>DFP Round To FP Integer With Inexact Quad Z23-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 20 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subset</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmm[ ]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>221</td>
<td>DFP Round Z23-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dmm[ ]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>221</td>
<td>DFP Round Quad Z23-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ddis[ ]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>230</td>
<td>DFP Round To DFP Short X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ddis[ ]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>238</td>
<td>DFP Shift Significant Left Immediate Z22-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ddis[ ]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>238</td>
<td>DFP Shift Significant Left Immediate Quad Z22-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dissect</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>213</td>
<td>DFP Test Data Class Z22-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dissectd</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>213</td>
<td>DFP Test Data Class Quad Z22-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>distdgg</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>213</td>
<td>DFP Test Data Group Quad Z22-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>distex</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>214</td>
<td>DFP Test Exponent X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>distexq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>214</td>
<td>DFP Test Exponent Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>distof</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>215</td>
<td>DFP Test Significance X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>distofq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>215</td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtxex</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Extract Biased Exponent X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtxexq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Extract Biased Exponent Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fcpesgn</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>161</td>
<td>Floating Copy Sign X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lbzcx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1164</td>
<td>Load Byte &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ldecx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1164</td>
<td>Load Doubleword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lfdp</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>159</td>
<td>Load Floating-Point Double Pair DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lfdpx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>159</td>
<td>Load Floating-Point Double Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>llwax</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>153</td>
<td>Load Floating-Point as Integer Word Algebraic Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>llwczx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1164</td>
<td>Load Halfword &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lstczx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1164</td>
<td>Load Word &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>prtyd</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>104</td>
<td>Parity Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>prtyw</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>103</td>
<td>Parity Word X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbfene</td>
<td>v2.05</td>
<td>P          SR</td>
<td>1229</td>
<td>SLB Find Entry ESID X-form</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sitbcx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Byte Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stddcx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Doubleword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stldp</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>160</td>
<td>Store Floating-Point Double Pair DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stldpx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>160</td>
<td>Store Floating-Point Double Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stshcx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Halfword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stwcx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Word Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>isel</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>98</td>
<td>Integer Select A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvebx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>268</td>
<td>Load Vector Element Byte Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvehx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>269</td>
<td>Load Vector Element Halfword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvewx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>270</td>
<td>Load Vector Element Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvel</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>277</td>
<td>Load Vector for Shift Left Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvex</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>277</td>
<td>Load Vector for Shift Right Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvxl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>271</td>
<td>Load Vector Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvxx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>271</td>
<td>Load Vector Indexed Last X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfhvscr</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>496</td>
<td>Move From Vector Status and Control Register VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mttvscr</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>496</td>
<td>Move To Vector Status and Control Register VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvebx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>272</td>
<td>Store Vector Element Byte Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvehx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>273</td>
<td>Store Vector Element Halfword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvewx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>274</td>
<td>Store Vector Element Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>275</td>
<td>Store Vector Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvxl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>275</td>
<td>Store Vector Indexed Last X-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 21 of 30)

Appendix F. Power ISA Instruction Set Sorted by Version
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
<th>Page Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>tielv</td>
<td>3212</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>64</td>
<td></td>
<td></td>
<td>1236</td>
<td>TLB Invalidate Entry Local X-form</td>
<td></td>
</tr>
<tr>
<td>vaddcqw</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>323</td>
<td></td>
<td></td>
<td>381</td>
<td>Vector Add &amp; write Carry Unsigned Word VX-form</td>
<td></td>
</tr>
<tr>
<td>vaddrp</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>422</td>
<td></td>
<td></td>
<td>380</td>
<td>Vector Add Floating-Point VX-form</td>
<td></td>
</tr>
<tr>
<td>vaddsbv</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>323</td>
<td></td>
<td></td>
<td>382</td>
<td>Vector Add Signed Byte Saturate VX-form</td>
<td></td>
</tr>
<tr>
<td>vaddshv</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>324</td>
<td></td>
<td></td>
<td>381</td>
<td>Vector Add Signed Halfword Saturate VX-form</td>
<td></td>
</tr>
<tr>
<td>vaddswm</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>325</td>
<td></td>
<td></td>
<td>380</td>
<td>Vector Add Signed Word Modulo VX-form</td>
<td></td>
</tr>
<tr>
<td>vaddshm</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>325</td>
<td></td>
<td></td>
<td>381</td>
<td>Vector Add Signed Halfword Modulo VX-form</td>
<td></td>
</tr>
<tr>
<td>vaddshv</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>327</td>
<td></td>
<td></td>
<td>380</td>
<td>Vector Add Signed Halfword VX-form</td>
<td></td>
</tr>
<tr>
<td>vaddswm</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>328</td>
<td></td>
<td></td>
<td>381</td>
<td>Vector Add Signed Word Modulo VX-form</td>
<td></td>
</tr>
<tr>
<td>vand</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>402</td>
<td></td>
<td></td>
<td>380</td>
<td>Vector Logical AND VX-form</td>
<td></td>
</tr>
<tr>
<td>vanic</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>402</td>
<td></td>
<td></td>
<td>381</td>
<td>Vector Logical AND with Complement VX-form</td>
<td></td>
</tr>
<tr>
<td>vavgib</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>375</td>
<td></td>
<td></td>
<td>382</td>
<td>Vector Average Signed Byte VX-form</td>
<td></td>
</tr>
<tr>
<td>vavgsh</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>376</td>
<td></td>
<td></td>
<td>383</td>
<td>Vector Average Signed Halfword VX-form</td>
<td></td>
</tr>
<tr>
<td>vavgsw</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>377</td>
<td></td>
<td></td>
<td>384</td>
<td>Vector Average Signed Word VX-form</td>
<td></td>
</tr>
<tr>
<td>vavgub</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>375</td>
<td></td>
<td></td>
<td>385</td>
<td>Vector Average Unsigned Byte VX-form</td>
<td></td>
</tr>
<tr>
<td>vavguh</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>376</td>
<td></td>
<td></td>
<td>386</td>
<td>Vector Average Signed Halfword VX-form</td>
<td></td>
</tr>
<tr>
<td>vavguw</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>377</td>
<td></td>
<td></td>
<td>387</td>
<td>Vector Average Unsigned Word VX-form</td>
<td></td>
</tr>
<tr>
<td>vcfux</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>425</td>
<td></td>
<td></td>
<td>388</td>
<td>Vector Convert with round to nearest From Signed Word to floating-point format VX-form</td>
<td></td>
</tr>
<tr>
<td>vcfsx</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>425</td>
<td></td>
<td></td>
<td>389</td>
<td>Vector Convert with round to nearest From Signed Word to floating-point format VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpbf[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>429</td>
<td></td>
<td></td>
<td>390</td>
<td>Vector Compare Bounds Floating-Point VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpbf[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>430</td>
<td></td>
<td></td>
<td>391</td>
<td>Vector Compare Equal Floating-Point VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpse[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>388</td>
<td></td>
<td></td>
<td>392</td>
<td>Vector Compare Equal Signed Byte VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpqse[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>389</td>
<td></td>
<td></td>
<td>393</td>
<td>Vector Compare Equal Signed Halfword VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmppe[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>388</td>
<td></td>
<td></td>
<td>394</td>
<td>Vector Compare Equal Signed Word VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpge[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>430</td>
<td></td>
<td></td>
<td>395</td>
<td>Vector Compare Greater Than or Equal Floating-Point VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpgt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>431</td>
<td></td>
<td></td>
<td>396</td>
<td>Vector Compare Greater Than Floating-Point VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpgt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>393</td>
<td></td>
<td></td>
<td>397</td>
<td>Vector Compare Greater Than Signed Byte VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpgts[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>394</td>
<td></td>
<td></td>
<td>398</td>
<td>Vector Compare Greater Than Signed Halfword VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpgt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>395</td>
<td></td>
<td></td>
<td>399</td>
<td>Vector Compare Greater Than Signed Word VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpgt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>393</td>
<td></td>
<td></td>
<td>400</td>
<td>Vector Compare Greater Than Unsigned Byte VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpgt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>394</td>
<td></td>
<td></td>
<td>401</td>
<td>Vector Compare Greater Than Unsigned Halfword VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpgt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>395</td>
<td></td>
<td></td>
<td>402</td>
<td>Vector Compare Greater Than Unsigned Word VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmplt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>426</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Compare with round to zero from floating-point To Signed Word format Saturate VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmplt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>426</td>
<td></td>
<td></td>
<td>404</td>
<td>Vector Compare with round to zero from floating-point To Signed Word format Saturate VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmplt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>426</td>
<td></td>
<td></td>
<td>405</td>
<td>Vector Compare with round to zero from floating-point To Signed Word format Saturate VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmpgt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>432</td>
<td></td>
<td></td>
<td>406</td>
<td>Vector Log Base 2 Estimate Floating-Point VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmplt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>423</td>
<td></td>
<td></td>
<td>407</td>
<td>Vector Multiply-Add Floating-Point VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmplt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>424</td>
<td></td>
<td></td>
<td>408</td>
<td>Vector Maximum Floating-Point VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmplt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>380</td>
<td></td>
<td></td>
<td>409</td>
<td>Vector Maximum Signed Byte VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmplt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>381</td>
<td></td>
<td></td>
<td>410</td>
<td>Vector Maximum Signed Halfword VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmplt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>382</td>
<td></td>
<td></td>
<td>411</td>
<td>Vector Maximum Signed Word VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmplt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>380</td>
<td></td>
<td></td>
<td>412</td>
<td>Vector Maximum Unsigned Byte VX-form</td>
<td></td>
</tr>
<tr>
<td>vcmplt[lj]</td>
<td>3209</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>381</td>
<td></td>
<td></td>
<td>413</td>
<td>Vector Maximum Unsigned Halfword VX-form</td>
<td></td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 22 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmaxuw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>382</td>
</tr>
<tr>
<td>vmladduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>352</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>290</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>291</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>290</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>291</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>290</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>291</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>290</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>291</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>290</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>291</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>290</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>291</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>290</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>291</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>290</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>291</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>290</td>
</tr>
<tr>
<td>vmladdduhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td>291</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 23 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsubcuw</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>331</td>
<td>Vector Subtract &amp; Write Carry-out Unsigned Word VX-form</td>
</tr>
<tr>
<td>vsubip</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>422</td>
<td>Vector Subtract Floating-Point VX-form</td>
</tr>
<tr>
<td>vsubsbs</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>331</td>
<td>Vector Subtract Signed Byte Saturate VX-form</td>
</tr>
<tr>
<td>vsubshs</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>332</td>
<td>Vector Subtract Signed Halfword Saturate VX-form</td>
</tr>
<tr>
<td>vsubsws</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>332</td>
<td>Vector Subtract Signed Word Saturate VX-form</td>
</tr>
<tr>
<td>vsubfwm</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>333</td>
<td>Vector Subtract Unaligned Halfword Modulo VX-form</td>
</tr>
<tr>
<td>vsubfrs</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>333</td>
<td>Vector Subtract Unaligned Word Modulo VX-form</td>
</tr>
<tr>
<td>vsubuwm</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>334</td>
<td>Vector Subtract Unaligned Word Modulo VX-form</td>
</tr>
<tr>
<td>vsubuws</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>335</td>
<td>Vector Subtract Unaligned Word Saturate VX-form</td>
</tr>
<tr>
<td>vsum2aws</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>336</td>
<td>Vector Subtract Unaligned Word Saturate VX-form</td>
</tr>
<tr>
<td>vsum4sbs</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>368</td>
<td>Vector Sum across Half Signed Word Saturate VX-form</td>
</tr>
<tr>
<td>vsum4shs</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>369</td>
<td>Vector Sum across Quarter Signed Byte Saturate VX-form</td>
</tr>
<tr>
<td>vsum4shs</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>369</td>
<td>Vector Sum across Quarter Signed Halfword Saturate VX-form</td>
</tr>
<tr>
<td>vsum4ubs</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>370</td>
<td>Vector Sum across Quarter Signed Byte Saturate VX-form</td>
</tr>
<tr>
<td>vsumsows</td>
<td>vx</td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td>VX-form</td>
<td></td>
<td></td>
<td>367</td>
<td>Vector Sum across Signed Word Saturate VX-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 24 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupkpx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>288</td>
<td>Vector Unpack High Doubleword to Double-Precision format X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupkhpsb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>285</td>
<td>Vector Unpack High Signed Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupkhsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Unpack High Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupkpx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>288</td>
<td>Vector Unpack Low Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupkshb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>285</td>
<td>Vector Unpack Low Signed Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupksh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Unpack Low Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupkipx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>288</td>
<td>Vector Unpack Low Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupklsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Unpack Low Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxor</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical XOR VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fdivr</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>165</td>
<td>Floating Reciprocal Estimate A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fdivr</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>176</td>
<td>Floating Round to Integer Minus X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>frnd</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>176</td>
<td>Floating Round to Integer Nearest X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fdivr</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>176</td>
<td>Floating Round to Integer Plus X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fdivr</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>176</td>
<td>Floating Round to Integer Toward Zero X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>frnd</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>165</td>
<td>Floating Reciprocal Square Root Estimate Single A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>frnd</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>152</td>
<td>Return From Interrupt Doubleword Hypervisor XL-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fdivr</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>103</td>
<td>Population Counts Bytes X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mlfcrf</td>
<td>v2.01</td>
<td></td>
<td></td>
<td>128</td>
<td>Move From One Condition Register Field XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mlfcfr</td>
<td>v2.01</td>
<td></td>
<td></td>
<td>127</td>
<td>Move To One Condition Register Field XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbmfcl</td>
<td>v2.00</td>
<td>P</td>
<td></td>
<td>1229</td>
<td>SLB Move From Entry ESID X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbmfcl</td>
<td>v2.00</td>
<td>P</td>
<td></td>
<td>1228</td>
<td>SLB Move From Entry VSID X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbrtf</td>
<td>v2.00</td>
<td>P</td>
<td></td>
<td>1227</td>
<td>SLB Move To Entry X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rfsclv</td>
<td>v3.0</td>
<td>P</td>
<td></td>
<td>1151</td>
<td>Return From System Call Vectored XL-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>scv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>47</td>
<td>System Call Vectored SC-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>liq</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>66</td>
<td>Store Quadword DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stq</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>66</td>
<td>Store Quadword DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>crlizd</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>104</td>
<td>Count Leading Zeros Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dbcf</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1064</td>
<td>Data Cache Block Flush X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dbct</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1063</td>
<td>Data Cache Block Store X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dbct</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1061</td>
<td>Data Cache Block Touch X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dbctb</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1062</td>
<td>Data Cache Block Touch for Store X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>divd</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>89</td>
<td>Divide Doubleword XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>divd</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>89</td>
<td>Divide Doubleword &amp; record OV XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>divd</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>89</td>
<td>Divide Doubleword Unsigned XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>divd</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>89</td>
<td>Divide Doubleword &amp; record OV XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>divd</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>82</td>
<td>Divide Word XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>divd</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>82</td>
<td>Divide Word &amp; record OV XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>divd</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>82</td>
<td>Divide Word Unsigned XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>divd</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>82</td>
<td>Divide Word &amp; record OV XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>divd</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>1068</td>
<td>Enforce In-order Execution of I/O X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>eextb</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>104</td>
<td>Extend Sign Byte X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>eextw</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>104</td>
<td>Extend Sign Word X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>eaddb</td>
<td>PPC</td>
<td></td>
<td></td>
<td>163</td>
<td>Floating Add Single A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fclid</td>
<td>PPC</td>
<td></td>
<td></td>
<td>174</td>
<td>Floating Convert with Signed Doubleword to Double-Precision format X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fclid</td>
<td>PPC</td>
<td></td>
<td></td>
<td>170</td>
<td>Floating Convert with Double-Precision To Signed Doubleword format X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fclid</td>
<td>PPC</td>
<td></td>
<td></td>
<td>171</td>
<td>Floating Convert with Double-Precision To Signed Doubleword format X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fclid</td>
<td>PPC</td>
<td></td>
<td></td>
<td>164</td>
<td>Floating Divide Single A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fmsadds</td>
<td>PPC</td>
<td></td>
<td></td>
<td>168</td>
<td>Floating Multiply-Add Single A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fmsadds</td>
<td>PPC</td>
<td></td>
<td></td>
<td>168</td>
<td>Floating Multiply-Add Single A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fmssubs</td>
<td>PPC</td>
<td></td>
<td></td>
<td>168</td>
<td>Floating Multiply-Subtract Single A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fmsubs</td>
<td>PPC</td>
<td></td>
<td></td>
<td>164</td>
<td>Floating Multiply Single A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fnmadds</td>
<td>PPC</td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Add Single A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fnmadds</td>
<td>PPC</td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Add Single A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fnmsubs</td>
<td>PPC</td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Subtract Single A-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 25 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>fres[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>165</td>
<td>Floating Reciprocal Estimate Single A-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>fnsqrt[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>166</td>
<td>Floating Reciprocal Square Root Estimate A-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>fsub[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>163</td>
<td>Floating Subtract A-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>icbi</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>1052</td>
<td>Instruction Cache Block Invalidate X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ld</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>180</td>
<td>Load Doubleword DS-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>tdi</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>1082</td>
<td>Load Doubleword And Reserve Indexed X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>tdu</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>157</td>
<td>Load Doubleword with Update DS-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>tfd</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>157</td>
<td>Load Doubleword with Update Indexed X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>tfd</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>157</td>
<td>Load Doubleword Indexed X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>wa</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>156</td>
<td>Load Word Algebraic DS-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>waux</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>1078</td>
<td>Load Word &amp; Reserve Indexed X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>lwax</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>56</td>
<td>Load Word Algebraic with Update Indexed X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>lwax</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>56</td>
<td>Load Word Algebraic Indexed X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>mtfb</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>1094</td>
<td>Move From Time Base FX-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>inmsrd</td>
<td>PPC</td>
<td>P</td>
<td>MD-form</td>
<td>1175</td>
<td>Move To MSR Doubleword X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>mulh[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>87</td>
<td>Multiply High Doubleword XO-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>mulhdu[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>87</td>
<td>Multiply High Doubleword Unsigned XO-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>multif[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>87</td>
<td>Multiply Low Doubleword XO-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>multil[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>87</td>
<td>Multiply Low Doubleword X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>multido[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>87</td>
<td>Multiply Low Doubleword &amp; record OV XO-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>rld[]</td>
<td>PPC</td>
<td>P</td>
<td>MD-form</td>
<td>1152</td>
<td>Return from Interrupt Doubleword XL-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ridc[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>111</td>
<td>Rotate Left Doubleword then Clear Left MD-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ridc[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>111</td>
<td>Rotate Left Doubleword then Clear Right MD-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ridc[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>110</td>
<td>Rotate Left Doubleword Immediate then Clear Left MD-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ridc[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>110</td>
<td>Rotate Left Doubleword Immediate then Clear Right MD-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ridc[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>112</td>
<td>Rotate Left Doubleword Immediate then Mask Insert MD-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sc</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>47</td>
<td>System Call SC-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sbia</td>
<td>PPC</td>
<td>P</td>
<td>MD-form</td>
<td>1224</td>
<td>SLB Invalidate All X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sbie</td>
<td>PPC</td>
<td>P</td>
<td>MD-form</td>
<td>1221</td>
<td>SLB Invalidate Entry X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sfd[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>115</td>
<td>Shift Left Doubleword X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>srd[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>115</td>
<td>Shift Right Algebraic Doubleword X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>stdx</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>115</td>
<td>Shift Right Algebraic Doubleword Immediate XS-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>stdx</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>115</td>
<td>Shift Right Doubleword X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>std[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>62</td>
<td>Store Doubleword DS-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>stdc[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>1082</td>
<td>Store Doubleword Conditional Indexed X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>std[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>62</td>
<td>Store Doubleword with Update DS-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>stdux</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>63</td>
<td>Store Doubleword with Update Indexed X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>stdx</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>62</td>
<td>Store Doubleword Indexed X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>stiwx</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>158</td>
<td>Store Floating-Point as Integer Word Indexed X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>stwcx</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>1081</td>
<td>Store Word Conditional Indexed X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>subf[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>77</td>
<td>Subtract From X-as Integer Word X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>std[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>98</td>
<td>Trap Doubleword X-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>std[]</td>
<td>PPC</td>
<td>SR</td>
<td>MD-form</td>
<td>98</td>
<td>Trap Doubleword Immediate D-form</td>
</tr>
<tr>
<td>011111......</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>tlbysync</td>
<td>PPC</td>
<td>HV/P</td>
<td>MD-form</td>
<td>1240</td>
<td>TLB Synchronize X-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 26 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliant/ Subsets</th>
<th>Linux Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>fctiw[.] P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>172</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fctiwz[.] P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>173</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f32r[.] P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>165</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>77</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addc[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>78</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addc[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>78</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addco[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>78</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>76</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>79</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>79</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>79</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>41</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>102</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>102</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>161</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>163</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>179</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>179</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>164</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>166</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>166</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[.] P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>164</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 27 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Complimentary Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fnabs[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>161</td>
<td>Floating Negative Absolute Value X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fneg[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>161</td>
<td>Floating Negative X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fnmadd[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Add A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fnmsub[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Subtract A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fnpl[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>170</td>
<td>Floating Round to Single-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fs subj[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>163</td>
<td>Floating Subtract A-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sysync</td>
<td>P1</td>
<td></td>
<td></td>
<td>1076</td>
<td>Instruction Synchronize XL-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lsz</td>
<td>P1</td>
<td></td>
<td></td>
<td>52</td>
<td>Load Byte and Zero D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lszu</td>
<td>P1</td>
<td></td>
<td></td>
<td>52</td>
<td>Load Byte and Zero with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lszux</td>
<td>P1</td>
<td></td>
<td></td>
<td>52</td>
<td>Load Byte and Zero with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lzx</td>
<td>P1</td>
<td></td>
<td></td>
<td>52</td>
<td>Load Byte and Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lzh</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td>Load Halfword and Zero D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lzhux</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td>Load Halfword and Zero with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhz</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td>Load Halfword and Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhzux</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td>Load Halfword and Zero with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhz</td>
<td>P1</td>
<td></td>
<td></td>
<td>70</td>
<td>Load Multiple Word D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwz</td>
<td>P1</td>
<td></td>
<td></td>
<td>72</td>
<td>Load String Word Immediate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwz</td>
<td>P1</td>
<td></td>
<td></td>
<td>72</td>
<td>Load String Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwz</td>
<td>P1</td>
<td></td>
<td></td>
<td>68</td>
<td>Load Word Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwu</td>
<td>P1</td>
<td></td>
<td></td>
<td>55</td>
<td>Load Word and Zero D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwu</td>
<td>P1</td>
<td></td>
<td></td>
<td>55</td>
<td>Load Word and Zero with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwux</td>
<td>P1</td>
<td></td>
<td></td>
<td>55</td>
<td>Load Word and Zero with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwz</td>
<td>P1</td>
<td></td>
<td></td>
<td>55</td>
<td>Load Word and Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwz</td>
<td>P1</td>
<td></td>
<td></td>
<td>46</td>
<td>Move Condition Register Field XL-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mcrf</td>
<td>P1</td>
<td></td>
<td></td>
<td>184</td>
<td>Move to Condition Register from FFSRC X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mcrf</td>
<td>P1</td>
<td></td>
<td></td>
<td>128</td>
<td>Move From Condition Register XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mifl[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>182</td>
<td>Move From FFSRC X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mifl[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>128</td>
<td>Move From Special Purpose Register XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mifl[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>127</td>
<td>Move To Condition Register Fields XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mifl[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>185</td>
<td>Move To FFSRC Bit 0 X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mifl[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>185</td>
<td>Move To FFSRC Bit 1 X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mifl[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>184</td>
<td>Move To FFSRC Fields XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mifl[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>184</td>
<td>Move To FFSRC Field Immediate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mifl[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>1174</td>
<td>Move To MSR X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mifl[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>124</td>
<td>Move To Special Purpose Register XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nulli</td>
<td>P1</td>
<td></td>
<td></td>
<td>81</td>
<td>Multi Low Immediate D-form</td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 28 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Privilege</th>
<th>Version</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulw[.] P1</td>
<td>SR</td>
<td>Version 3.1</td>
<td>81</td>
<td>Multiply Low Word XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulwc[.] P1</td>
<td>SR</td>
<td>Version 3.1</td>
<td>81</td>
<td>Multiply Low Word &amp; record OV XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nand[.] P1</td>
<td>SR</td>
<td>Version 100</td>
<td>100</td>
<td>NAND XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>neg[.] P1</td>
<td>SR</td>
<td>Version 80</td>
<td>80</td>
<td>Negate XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>negc[.] P1</td>
<td>SR</td>
<td>Version 80</td>
<td>80</td>
<td>Negate &amp; record OV XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nor[.] P1</td>
<td>SR</td>
<td>Version 101</td>
<td>101</td>
<td>NOR XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ori[.] P1</td>
<td>SR</td>
<td>Version 101</td>
<td>101</td>
<td>OR XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>orci[.] P1</td>
<td>SR</td>
<td>Version 101</td>
<td>101</td>
<td>OR with Complement X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or[.] P1</td>
<td>SR</td>
<td>Version 98</td>
<td>98</td>
<td>OR Immediate D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>orci[.] P1</td>
<td>SR</td>
<td>Version 100</td>
<td>100</td>
<td>OR Immediate Shifted D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rlwm[.] P1</td>
<td>SR</td>
<td>Version 106</td>
<td>106</td>
<td>Rotate Left Word Immediate then Mask Insert M-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rlwmi[.] P1</td>
<td>SR</td>
<td>Version 107</td>
<td>107</td>
<td>Rotate Left Word Immediate then AND with Mask M-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>siw[.] P1</td>
<td>SR</td>
<td>Version 113</td>
<td>113</td>
<td>Shift Left Word XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>siwl[.] P1</td>
<td>SR</td>
<td>Version 113</td>
<td>113</td>
<td>Shift Right Algebraic Word X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>siwl[.] P1</td>
<td>SR</td>
<td>Version 113</td>
<td>113</td>
<td>Shift Right Algebraic Word Immediate X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>siwl[.] P1</td>
<td>SR</td>
<td>Version 113</td>
<td>113</td>
<td>Shift Right Word XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stib P1</td>
<td></td>
<td>Version 59</td>
<td>59</td>
<td>Store Byte D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stbu P1</td>
<td></td>
<td>Version 59</td>
<td>59</td>
<td>Store Byte with Update D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stbux P1</td>
<td></td>
<td>Version 59</td>
<td>59</td>
<td>Store Byte with Update Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stbx P1</td>
<td></td>
<td>Version 59</td>
<td>59</td>
<td>Store Byte Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stfs P1</td>
<td></td>
<td>Version 155</td>
<td>155</td>
<td>Store Floating-Point Single D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stfs P1</td>
<td></td>
<td>Version 155</td>
<td>155</td>
<td>Store Floating-Point Single Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stfsux P1</td>
<td></td>
<td>Version 155</td>
<td>155</td>
<td>Store Floating-Point Single with Update Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stfsx P1</td>
<td></td>
<td>Version 155</td>
<td>155</td>
<td>Store Floating-Point Single Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stbx P1</td>
<td></td>
<td>Version 60</td>
<td>60</td>
<td>Store Byte D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stbux P1</td>
<td></td>
<td>Version 60</td>
<td>60</td>
<td>Store Byte with Update D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stbux P1</td>
<td></td>
<td>Version 60</td>
<td>60</td>
<td>Store Byte with Update Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stbx P1</td>
<td></td>
<td>Version 60</td>
<td>60</td>
<td>Store Byte Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stmw P1</td>
<td></td>
<td>Version 70</td>
<td>70</td>
<td>Store Multiple Word D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stsw P1</td>
<td></td>
<td>Version 73</td>
<td>73</td>
<td>Store String Word Immediate X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stswx P1</td>
<td></td>
<td>Version 73</td>
<td>73</td>
<td>Store String Word Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stsw P1</td>
<td></td>
<td>Version 61</td>
<td>61</td>
<td>Store Word D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stswbrx P1</td>
<td></td>
<td>Version 68</td>
<td>68</td>
<td>Store Word Byte-Reverse Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stsw P1</td>
<td></td>
<td>Version 61</td>
<td>61</td>
<td>Store Word with Update D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stsw P1</td>
<td></td>
<td>Version 61</td>
<td>61</td>
<td>Store Word Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stsw P1</td>
<td></td>
<td>Version 61</td>
<td>61</td>
<td>Store Word Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subc[.] P1</td>
<td>SR</td>
<td>Version 78</td>
<td>78</td>
<td>Subtract From Carrying XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subc[.] P1</td>
<td>SR</td>
<td>Version 78</td>
<td>78</td>
<td>Subtract From Carrying &amp; record OV XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subc[.] P1</td>
<td>SR</td>
<td>Version 78</td>
<td>78</td>
<td>Subtract From Extended XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subc[.] P1</td>
<td>SR</td>
<td>Version 78</td>
<td>78</td>
<td>Subtract From Extended &amp; record OV XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subc P1</td>
<td>SR</td>
<td>Version 77</td>
<td>77</td>
<td>Subtract From Immediate Carrying D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subc P1</td>
<td>SR</td>
<td>Version 79</td>
<td>79</td>
<td>Subtract From Minus One Extended XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subc P1</td>
<td>SR</td>
<td>Version 79</td>
<td>79</td>
<td>Subtract From Minus One Extended &amp; record OV XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subc P1</td>
<td>SR</td>
<td>Version 78</td>
<td>78</td>
<td>Subtract From Zero Extended XO-form</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 92. Power ISA AS Instruction Set Sorted by Version (Sheet 29 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111.......</td>
<td>P1</td>
<td>SR</td>
<td>78</td>
<td>Subtract From Zero Extended &amp; record OV XO-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111.......</td>
<td>P1</td>
<td>1086</td>
<td>Synchronize X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111.......</td>
<td>P1</td>
<td>64</td>
<td>TLB Invalidate Entry X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111.......</td>
<td>P1</td>
<td>97</td>
<td>Trap Word X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111.......</td>
<td>P1</td>
<td>97</td>
<td>Trap Word Immediate D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111.......</td>
<td>P1</td>
<td>100</td>
<td>XOR Immediate D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111.......</td>
<td>P1</td>
<td>100</td>
<td>XOR Immediate Shifted D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Instruction

1 Instruction bit that corresponds to a reserved field, must have a value of 0, otherwise invalid form.
0 Instruction bit that corresponds to an operand bit, may have a value of either 0 or 1.
1 Instruction bit that corresponds to an opcode bit having a value 0.
0 Instruction bit that corresponds to an opcode bit having a value 1.

2. OpenPOWER Compliancy Subsets

X... Instruction included in the Scalar Fixed-Point Compliancy subset
.X.. Instruction included in the Scalar Fixed-Point + Floating-Point Compliancy subset.
..X. Instruction included in the Linux Compliancy subset.
...X Instruction included in the AIX Compliancy subset.

3. Linux Optional Category

AMO Instruction part of Atomic Memory Operations category.
BFP128 Instruction part of Quad-Precision Floating-Point category.
BHRB Instruction part of Branch History Rolling Buffer category.
DFP Instruction part of Decimal Floating-Point category.
EBB Instruction part of Event-Based Branch category.
MMA Instruction part of Matrix-Multiplication Assist category.

4. Always Optional Category

MMA Instruction part of Matrix-Multiplication Assist category.

5. Version

P1 Instruction introduced in POWER Architecture.
P2 Instruction introduced in POWER2 Architecture.
PPC Instruction introduced in PowerPC Architecture prior to v2.00.
v2.00 Instruction introduced in PowerPC Architecture Version 2.00.
v2.01 Instruction introduced in PowerPC Architecture Version 2.01.
v2.02 Instruction introduced in PowerPC Architecture Version 2.02.
v2.03 Instruction introduced in Power ISA Version 2.03.
v2.04 Instruction introduced in Power ISA Version 2.04.
v2.05 Instruction introduced in Power ISA Version 2.05.
v2.06 Instruction introduced in Power ISA Version 2.06.
v2.07 Instruction introduced in Power ISA Version 2.07.
v3.0 Instruction introduced in Power ISA Version 3.0.
v3.0B Instruction introduced in Power ISA Version 3.0B.
v3.0C Instruction introduced in Power ISA Version 3.0C.
v3.1 Instruction introduced in Power ISA Version 3.1.

6. Privilege

P Denotes an instruction that is treated as privileged.
O Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor-privileged for mtspr), depending on the SPR or PMR number.
PI Denotes an instruction that is illegal in privileged state.
HV Denotes an instruction that can be executed only in hypervisor state.
UV Denotes an instruction that can be executed only in ultravisor state.
7. **Mode Dependency.**

Except as described below and in Section 1.10.3, "Effective Address Calculation", in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

- **CT** If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
- **SR** The setting of status registers (such as XER and CR0) is mode-dependent.
- **SF=1** The instruction can be executed only in 64-bit mode.
Appendix G. Power ISA Instruction Set Sorted by OpenPOWER Compliancy Subset

This appendix lists all the instructions in the Power ISA, sorted by the instruction’s OpenPOWER compliancy subset membership, then by mnemonic.

<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Book</th>
<th>Compliancy Subsets²</th>
<th>Linux Optional Category³</th>
<th>Always Optional Category⁴</th>
<th>Mnemonic</th>
<th>Version⁵</th>
<th>Privilege⁶</th>
<th>Mode Dep⁷</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 1 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>crandc</td>
<td>P1</td>
<td></td>
<td></td>
<td>45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>creqv</td>
<td>P1</td>
<td></td>
<td></td>
<td>45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmdand</td>
<td>P1</td>
<td></td>
<td></td>
<td>44</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmor</td>
<td>P1</td>
<td></td>
<td></td>
<td>45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>crnor</td>
<td>P1</td>
<td></td>
<td></td>
<td>45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>crorc</td>
<td>P1</td>
<td></td>
<td></td>
<td>45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>eqv</td>
<td>P1</td>
<td></td>
<td></td>
<td>45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>extsb</td>
<td>PPC</td>
<td>SR</td>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>extsh</td>
<td>P1</td>
<td></td>
<td></td>
<td>45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lbz</td>
<td>P1</td>
<td></td>
<td></td>
<td>52</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lbzu</td>
<td>P1</td>
<td></td>
<td></td>
<td>52</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lbzx</td>
<td>P1</td>
<td></td>
<td></td>
<td>52</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lha</td>
<td>P1</td>
<td></td>
<td></td>
<td>54</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhau</td>
<td>P1</td>
<td></td>
<td></td>
<td>54</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhax</td>
<td>P1</td>
<td></td>
<td></td>
<td>54</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhz</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhzu</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhux</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhx</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwbx</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwz</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwu</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwux</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mcr</td>
<td>P1</td>
<td></td>
<td></td>
<td>46</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mcrf</td>
<td>P1</td>
<td></td>
<td></td>
<td>46</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfr</td>
<td>P1</td>
<td></td>
<td></td>
<td>46</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfmsr</td>
<td>P1</td>
<td></td>
<td></td>
<td>4176</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfsr</td>
<td>P1</td>
<td></td>
<td></td>
<td>4176</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mulh</td>
<td>PPC</td>
<td>SR</td>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mulhwu</td>
<td>PPC</td>
<td>SR</td>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mulwi</td>
<td>P1</td>
<td></td>
<td></td>
<td>81</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>muluw</td>
<td>P1</td>
<td></td>
<td></td>
<td>81</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nand</td>
<td>P1</td>
<td></td>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>neg</td>
<td>P1</td>
<td></td>
<td></td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 2 of 30)
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Book</th>
<th>Compliancy Subsets2</th>
<th>Linux Optional Category3</th>
<th>Always Optional Category4</th>
<th>Mnemonic</th>
<th>Version5</th>
<th>Privilege6</th>
<th>Mode Dep7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>nor[i]</td>
<td>P1</td>
<td>SR</td>
<td>101</td>
<td>NOR X-form</td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>or[i]</td>
<td>P1</td>
<td>SR</td>
<td>101</td>
<td>OR X-form</td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>orc[i]</td>
<td>P1</td>
<td>99</td>
<td>OR Immediate D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>oris[i]</td>
<td>P1</td>
<td>100</td>
<td>OR Immediate Shifted D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>popcntb</td>
<td>v2.02</td>
<td>103</td>
<td>Population Count Bytes X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>popcntw</td>
<td>v2.06</td>
<td>103</td>
<td>Population Count Words X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>popcntw</td>
<td>v2.05</td>
<td>103</td>
<td>Parity Word X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>riwmi[i]</td>
<td>P1</td>
<td>SR</td>
<td>106</td>
<td>Rotate Left Word Immediate then Mask Insert M-form</td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>riwmi[i]</td>
<td>P1</td>
<td>SR</td>
<td>107</td>
<td>Rotate Left Word Immediate then AND with Mask M-form</td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>sc</td>
<td>PPC</td>
<td>47</td>
<td>System Call SC-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>scv</td>
<td>v3.0</td>
<td>47</td>
<td>System Call Vectored SC-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>setb</td>
<td>v3.0</td>
<td>129</td>
<td>Set Boolean X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>setbcr</td>
<td>v3.1</td>
<td>129</td>
<td>Set Boolean Condition Reverse X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>setbnc</td>
<td>v3.1</td>
<td>129</td>
<td>Set Negative Boolean Condition X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>setbncr</td>
<td>v3.1</td>
<td>129</td>
<td>Set Negative Boolean Condition Reverse X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>sib[i]</td>
<td>P1</td>
<td>SR</td>
<td>113</td>
<td>Shift Left Word X-form</td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>sibu[i]</td>
<td>P1</td>
<td>59</td>
<td>Store Byte D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>sibux[i]</td>
<td>P1</td>
<td>59</td>
<td>Store Byte with Update D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>stb[i]</td>
<td>P1</td>
<td>59</td>
<td>Store Byte Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>stbu[i]</td>
<td>P1</td>
<td>60</td>
<td>Store Halfword D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>stbx[i]</td>
<td>P1</td>
<td>60</td>
<td>Store Halfword with Update D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>stbx[i]</td>
<td>P1</td>
<td>60</td>
<td>Store Halfword Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>stw[i]</td>
<td>P1</td>
<td>61</td>
<td>Store Word D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>stwx[i]</td>
<td>P1</td>
<td>61</td>
<td>Store Word with Update D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>stwux[i]</td>
<td>P1</td>
<td>61</td>
<td>Store Word with Update Indexed X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>sub[i]</td>
<td>PPC</td>
<td>77</td>
<td>Subtract From XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>subc[i]</td>
<td>P1</td>
<td>SR</td>
<td>76</td>
<td>Subtract From Carrying XO-form</td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>subfe[i]</td>
<td>P1</td>
<td>SR</td>
<td>75</td>
<td>Subtract From Extended XO-form</td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>subfc[i]</td>
<td>P1</td>
<td>SR</td>
<td>77</td>
<td>Subtract From Immediate Carrying D-form</td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>subfme[i]</td>
<td>P1</td>
<td>SR</td>
<td>79</td>
<td>Subtract From Minus One Extended XO-form</td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>subfze[i]</td>
<td>P1</td>
<td>SR</td>
<td>78</td>
<td>Subtract From Zero Extended XO-form</td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>tw</td>
<td>P1</td>
<td>97</td>
<td>Trap Word X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>twi[i]</td>
<td>P1</td>
<td>97</td>
<td>Trap Word Immediate D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>xor[i]</td>
<td>P1</td>
<td>SR</td>
<td>100</td>
<td>XOR X-form</td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>xor[i]</td>
<td>P1</td>
<td>100</td>
<td>XOR Immediate D-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>fabs[i]</td>
<td>P1</td>
<td>161</td>
<td>Floating Absolute Value X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>fadd[i]</td>
<td>P1</td>
<td>163</td>
<td>Floating Add A-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111111111</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>fadd[i]</td>
<td>PPC</td>
<td>163</td>
<td>Floating Add Single A-form</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 3 of 30)
<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Book</th>
<th>Compliancy Subsets 2</th>
<th>Linux Optional Category 3</th>
<th>Always Optional Category 4</th>
<th>Mnemonic</th>
<th>Version 5</th>
<th>Privilege 6</th>
<th>Mode Dep 7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fcid[.]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>174</td>
<td>Floating Convert with round Signed Doubleword to Double-Precision format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fchts[.]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>175</td>
<td>Floating Convert with round Signed Doubleword to Single-Precision format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fchdu[.]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>175</td>
<td>Floating Convert with round Unsigned Doubleword to Double-Precision format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fchdus[.]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>176</td>
<td>Floating Convert with round Unsigned Doubleword to Single-Precision format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fcmpo</td>
<td>P1</td>
<td></td>
<td></td>
<td>179</td>
<td>Floating Compare Ordered X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fcmpu</td>
<td>P2</td>
<td></td>
<td></td>
<td>179</td>
<td>Floating Compare Unordered X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fcpshr[.]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>161</td>
<td>Floating Copy Sign X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fctid[.]</td>
<td>PPC</td>
<td></td>
<td></td>
<td>170</td>
<td>Floating Convert with round Double-Precision To Signed Doubleword format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fctidu[.]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>171</td>
<td>Floating Convert with round Double-Precision To Signed Doubleword format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fctiduz[.</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>172</td>
<td>Floating Convert with truncate Double-Precision To Signed Doubleword format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fctidz[.</td>
<td>PPC</td>
<td></td>
<td></td>
<td>171</td>
<td>Floating Convert with truncate Double-Precision To Signed Doubleword format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fctiwv[.</td>
<td>P2</td>
<td></td>
<td></td>
<td>172</td>
<td>Floating Convert with round Double-Precision To Signed Word format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fctiw[.]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>173</td>
<td>Floating Convert with round Double-Precision To Signed Word format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fctiw[.]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>173</td>
<td>Floating Convert with round Double-Precision To Signed Word format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fctiw[.</td>
<td>P2</td>
<td></td>
<td></td>
<td>173</td>
<td>Floating Convert with round Double-Precision To Signed Word format X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fdiv[.]</td>
<td>P1</td>
<td></td>
<td></td>
<td>164</td>
<td>Floating Divide A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fdiv[.</td>
<td>PPC</td>
<td></td>
<td></td>
<td>164</td>
<td>Floating Divide Single-A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fmaadd[.</td>
<td>P1</td>
<td></td>
<td></td>
<td>168</td>
<td>Floating Multiply-Add A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fmaadds[.</td>
<td>PPC</td>
<td></td>
<td></td>
<td>168</td>
<td>Floating Multiply-Add Single-A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fmr[.]</td>
<td>P1</td>
<td></td>
<td></td>
<td>161</td>
<td>Floating Move Register X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnmrgw[.</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>162</td>
<td>Floating Merge Even Word X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnmrgw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>162</td>
<td>Floating Merge Odd Word X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnmsub[.</td>
<td>P1</td>
<td></td>
<td></td>
<td>168</td>
<td>Floating Multiply-Subtract A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnmsub[.</td>
<td>PPC</td>
<td></td>
<td></td>
<td>168</td>
<td>Floating Multiply-Subtract Single-A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnmulp[.</td>
<td>P1</td>
<td></td>
<td></td>
<td>164</td>
<td>Floating Multiply A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnmul[.</td>
<td>PPC</td>
<td></td>
<td></td>
<td>164</td>
<td>Floating Multiply Single-A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnabs[.</td>
<td>P1</td>
<td></td>
<td></td>
<td>161</td>
<td>Floating Negative Absolute Value X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fneg[.</td>
<td>P1</td>
<td></td>
<td></td>
<td>161</td>
<td>Floating Negate X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnmad[.</td>
<td>P1</td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Add A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnmad[.</td>
<td>PPC</td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Add Single-A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnmsub[.</td>
<td>P1</td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Subtract A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnmsub[.</td>
<td>PPC</td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Subtract Single-A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnneg[.</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>165</td>
<td>Floating Reciprocal Estimate A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnm[.</td>
<td>PPC</td>
<td></td>
<td></td>
<td>165</td>
<td>Floating Reciprocal Estimate Single-A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnm[.</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>178</td>
<td>Floating Round to Integer Minus X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnr[.</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>178</td>
<td>Floating Round to Integer Nearest X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnr[.</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>178</td>
<td>Floating Round to Integer Plus X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnzr[.</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>178</td>
<td>Floating Round to Integer Toward Zero X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnzr[.</td>
<td>P1</td>
<td></td>
<td></td>
<td>170</td>
<td>Floating Round to Single-Precision X-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnsqre[.</td>
<td>PPC</td>
<td></td>
<td></td>
<td>166</td>
<td>Floating Reciprocal Square Root Estimate A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fnsqre[.</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>166</td>
<td>Floating Reciprocal Square Root Estimate Single-A-form</td>
</tr>
<tr>
<td>111111111111</td>
<td>PPC</td>
<td>I</td>
<td></td>
<td></td>
<td>fns[.</td>
<td>PPC</td>
<td></td>
<td></td>
<td>180</td>
<td>Floating Select A-form</td>
</tr>
<tr>
<td>Instruction¹</td>
<td>Back</td>
<td>Compliancy Subsets²</td>
<td>Linux Optional Category³</td>
<td>Always Optional Category⁴</td>
<td>Mnemonic</td>
<td>Version⁵</td>
<td>Privilege⁶</td>
<td>Mode Dep⁷</td>
<td>Page</td>
<td>Name</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
<td>----------------------</td>
<td>--------------------------</td>
<td>---------------------------</td>
<td>----------</td>
<td>----------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 5 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 6 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Compliancy Subset</th>
<th>Linux Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxsrd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>610</td>
<td>Load VSX Scalar Doubleword DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxsdx</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>611</td>
<td>Load VSX Scalar Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxsibxx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>612</td>
<td>Load VSX Scalar as Integer Byte &amp; Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxsiwax</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>613</td>
<td>Load VSX Scalar as Integer Word Algebraic Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lsssp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>614</td>
<td>Load VSX Scalar Single-Precision DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lssspx</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>615</td>
<td>Load VSX Scalar Single-Precision Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>616</td>
<td>Load VSX Vector DQ-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvbf16x</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>617</td>
<td>Load VSX Vector Byte16 Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvd2x</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>618</td>
<td>Load VSX Vector Doubleword*2 Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvedx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>619</td>
<td>Load VSX Vector Doubleword &amp; Splat Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvfhdx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>620</td>
<td>Load VSX Vector with Length X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvfl</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>621</td>
<td>Load VSX Vector with Length Left-justified X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvld</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>622</td>
<td>Load VSX Vector Packed Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvldx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>623</td>
<td>Load VSX Vector Rightmost Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvpl</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>624</td>
<td>Load VSX Vector Rightmost Halfword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvw4x</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>625</td>
<td>Load VSX Vector Doubleword*4 Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvw8x</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>626</td>
<td>Load VSX Vector Word*8 Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxw4x</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>627</td>
<td>Load VSX Vector Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mxaddhu</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>628</td>
<td>Load VSX Vector Unaligned Doubleword VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mxaddid</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>629</td>
<td>Load VSX Vector Rightmost Halfword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mxw4x</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>630</td>
<td>Load VSX Vector Rightmost Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mxw8x</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>631</td>
<td>Load VSX Vector Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mxv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>632</td>
<td>Load VSX Vector Indexed DQ-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>madd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>633</td>
<td>Load VSX Vector Doubleword &amp; Splat Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>maddid</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>634</td>
<td>Load VSX Vector Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mulvcr</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>635</td>
<td>Move From Vector Status and Control Register VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mvfcrd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>636</td>
<td>Move From VSX Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mvfcrdl</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>637</td>
<td>Move From VSX Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mvfsnwc</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>638</td>
<td>Move From VSX Word and Zero X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mbssd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>639</td>
<td>Move To VSX Quadword Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mbssdl</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>640</td>
<td>Move To VSX Quadword Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mvfsnw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>641</td>
<td>Move To VSX Word Algebraic X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mvfnrm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>642</td>
<td>Move To VSX Word and Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvfrtm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>643</td>
<td>Move To VSX Byte Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvsrbsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>644</td>
<td>Move To VSX Byte Mask Immediate DX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvfrtm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>645</td>
<td>Move To VSX Byte Mask Immediate DX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvfrtd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>646</td>
<td>Move To VSX Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvfrtdd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>647</td>
<td>Move To VSX Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvfrth</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>648</td>
<td>Move To VSX Word Algebraic X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvfrtnw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>649</td>
<td>Move To VSX Word Mask VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvfrtn</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>650</td>
<td>Move To VSX Word &amp; Splat Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvfrtnw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>651</td>
<td>Move To VSX Word and Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvfrtl</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>652</td>
<td>Move To VSX High Doubleword XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvfrtlu</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>653</td>
<td>Move To VSX High Doubleword Unaligned XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mtvfrtdl</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>654</td>
<td>Move To VSX Low Doubleword XO-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 7 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets2</th>
<th>Linux Optional Category3</th>
<th>Always Optional Category4</th>
<th>Mnemonic</th>
<th>Version5</th>
<th>Privilege6</th>
<th>Mode Dep7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mulldo</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>87</td>
<td>Multiply Low Doubleword &amp; record OV XO-form</td>
</tr>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mullwo</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>81</td>
<td>Multiply Low Word &amp; record OV XO-form</td>
</tr>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nego</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>80</td>
<td>Negate &amp; record OV XO-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>paddi</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>76</td>
<td>Prefixed Add Immediate MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>paste</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>1068</td>
<td>Paste X-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pdepd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>106</td>
<td>Parallel Bits Deposit Doubleword X-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pextd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>106</td>
<td>Parallel Bits Extract Doubleword X-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plbz</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>52</td>
<td>Prefixed Load Byte and Zero MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plld</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>57</td>
<td>Prefixed Load Doubleword 8LS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plfd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>152</td>
<td>Prefixed Load Floating-Point Double MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pls</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>150</td>
<td>Prefixed Load Floating-Point Single MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plha</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>54</td>
<td>Prefixed Load Halfword Algebraic MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plhz</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>53</td>
<td>Prefixed Load Halfword and Zero MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>65</td>
<td>Prefixed Load Quadword 8LS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plwa</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>58</td>
<td>Prefixed Load Word Algebraic 8LS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plwz</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>55</td>
<td>Prefixed Load Word and Zero MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plxsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>610</td>
<td>Prefixed Load VSX Scalar Doubleword 8LS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plsssp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>615</td>
<td>Prefixed Load VSX Scalar Single-Precision 8LS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plkv</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>617</td>
<td>Prefixed Load VSX Vector 8LS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plxvp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>625</td>
<td>Prefixed Load VSX Vector Packed 8LS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pnonp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>130</td>
<td>Prefixed Nop MRR:*-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>popcntld</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>104</td>
<td>Population Count Doubleword X-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ptryd</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>104</td>
<td>Parity Doubleword X-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pstsrb</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>59</td>
<td>Prefixed Store Byte MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pstd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>62</td>
<td>Prefixed Store Doubleword 8LS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pstdt</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>157</td>
<td>Prefixed Store Floating-Point Double MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pstsfs</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>155</td>
<td>Prefixed Store Floating-Point Single MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>psth</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>60</td>
<td>Prefixed Store Halfword MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pstq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>66</td>
<td>Prefixed Store Quadword 8LS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pstrw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>61</td>
<td>Prefixed Store Word MLS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pstxsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>638</td>
<td>Prefixed Store VSX Scalar Doubleword 8LS:D-form</td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pstxssp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>642</td>
<td>Prefixed Store VSX Scalar Single-Precision 8LS:D-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 8 of 30)
<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Book</th>
<th>Compliancy Subsets 2</th>
<th>Linux Optional Category 3</th>
<th>Always Optional Category 4</th>
<th>Mnemonic</th>
<th>Version 5</th>
<th>Privilege 6</th>
<th>Mode Dep 7</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 9 of 30)
<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Book</th>
<th>Compliancy Subsets 2</th>
<th>Linux Optional Category 3</th>
<th>Always Optional Category 4</th>
<th>Mnemonic</th>
<th>Version 5</th>
<th>Privilege 6</th>
<th>Mode Dep 7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbvx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>656</td>
<td>Store VSX Vector Indexed X-form</td>
</tr>
<tr>
<td>subco[ ]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>78</td>
<td>Subtract From Carrying &amp; record OV XO-form</td>
</tr>
<tr>
<td>subtlco[ ]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>78</td>
<td>Subtract From Extended &amp; record OV XO-form</td>
</tr>
<tr>
<td>subfmeo[ ]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>79</td>
<td>Subtract From Minus One Extended &amp; record OV XO-form</td>
</tr>
<tr>
<td>subfo[ ]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>77</td>
<td>Subtract From &amp; record OV XO-form</td>
</tr>
<tr>
<td>sync</td>
<td>P1</td>
<td>1086</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Synchronize X-form</td>
</tr>
<tr>
<td>tdi</td>
<td>PPC</td>
<td>98</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Trap Doubleword X-form</td>
</tr>
<tr>
<td>vabsdub</td>
<td>v3.0</td>
<td>378</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Absolute Difference Unsigned Byte VX-form</td>
</tr>
<tr>
<td>vabsduh</td>
<td>v3.0</td>
<td>378</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Absolute Difference Unsigned Halfword VX-form</td>
</tr>
<tr>
<td>vabsduw</td>
<td>v3.0</td>
<td>379</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Absolute Difference Unsigned Word VX-form</td>
</tr>
<tr>
<td>vaddcuw</td>
<td>v2.07</td>
<td>323</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add &amp; write Carry Unsigned Word VX-form</td>
</tr>
<tr>
<td>vaddcuq</td>
<td>v2.03</td>
<td>323</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add &amp; write Carry Unsigned Word VX-form</td>
</tr>
<tr>
<td>vadddecuq</td>
<td>v2.07</td>
<td>330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
</tr>
<tr>
<td>vadddeuqm</td>
<td>v2.07</td>
<td>329</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended Unsigned Quadword Modulo VX-form</td>
</tr>
<tr>
<td>vaddrb</td>
<td>v2.03</td>
<td>422</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Floating-Point VX-form</td>
</tr>
<tr>
<td>vaddslb</td>
<td>v2.03</td>
<td>323</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Signed Byte Saturate VX-form</td>
</tr>
<tr>
<td>vaddsbs</td>
<td>v2.03</td>
<td>324</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Signed Halfword Saturate VX-form</td>
</tr>
<tr>
<td>vaddsws</td>
<td>v2.03</td>
<td>324</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Signed Word Saturate VX-form</td>
</tr>
<tr>
<td>vaddudum</td>
<td>v2.03</td>
<td>325</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Unsigned Byte Modulo VX-form</td>
</tr>
<tr>
<td>vaddduubs</td>
<td>v2.03</td>
<td>327</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Unsigned Byte Saturate VX-form</td>
</tr>
<tr>
<td>vaddudm</td>
<td>v2.07</td>
<td>326</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Unsigned Doubleword Modulo VX-form</td>
</tr>
<tr>
<td>vadduuhm</td>
<td>v2.03</td>
<td>325</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Unsigned Halfword Modulo VX-form</td>
</tr>
<tr>
<td>vadduhs</td>
<td>v2.03</td>
<td>327</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Unsigned Halfword Saturate VX-form</td>
</tr>
<tr>
<td>vadduqm</td>
<td>v2.07</td>
<td>329</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Unsigned Quadword Modulo VX-form</td>
</tr>
<tr>
<td>vadduwm</td>
<td>v2.03</td>
<td>326</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Unsigned Word Modulo VX-form</td>
</tr>
<tr>
<td>vadduws</td>
<td>v2.03</td>
<td>328</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Unsigned Word Saturate VX-form</td>
</tr>
<tr>
<td>vadduux</td>
<td>v2.03</td>
<td>328</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Unsigned Word VX-form</td>
</tr>
<tr>
<td>vandc</td>
<td>v2.03</td>
<td>402</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Logical AND &amp; write Carry &amp; record OV XO-form</td>
</tr>
<tr>
<td>vaggb</td>
<td>v2.03</td>
<td>375</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Signed Byte VX-form</td>
</tr>
<tr>
<td>vaggh</td>
<td>v2.03</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Signed Halfword VX-form</td>
</tr>
<tr>
<td>vaggw</td>
<td>v2.03</td>
<td>377</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Signed Word VX-form</td>
</tr>
<tr>
<td>vabsduw</td>
<td>v2.03</td>
<td>375</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Signed Byte VX-form</td>
</tr>
<tr>
<td>vabsduh</td>
<td>v2.03</td>
<td>375</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Signed Halfword VX-form</td>
</tr>
<tr>
<td>vabsduw</td>
<td>v2.03</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Signed Word VX-form</td>
</tr>
<tr>
<td>vavgub</td>
<td>v2.03</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Signed Halfword VX-form</td>
</tr>
<tr>
<td>vavgul</td>
<td>v2.03</td>
<td>377</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Signed Word VX-form</td>
</tr>
<tr>
<td>vavgub</td>
<td>v2.03</td>
<td>375</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Signed Byte VX-form</td>
</tr>
<tr>
<td>vavgul</td>
<td>v2.03</td>
<td>375</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Signed Halfword VX-form</td>
</tr>
<tr>
<td>vavgub</td>
<td>v2.03</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Signed Word VX-form</td>
</tr>
<tr>
<td>vavguw</td>
<td>v2.03</td>
<td>377</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Average Unaligned Word VX-form</td>
</tr>
<tr>
<td>vbpermmd</td>
<td>v3.0</td>
<td>461</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Bit Permute Doubleword VX-form</td>
</tr>
<tr>
<td>vbpermq</td>
<td>v2.07</td>
<td>462</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Bit Permute Quadword VX-form</td>
</tr>
<tr>
<td>vcfx</td>
<td>v2.03</td>
<td>425</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Convert with round to nearest From Signed Word to floating-point format VX-form</td>
</tr>
<tr>
<td>vcfuges</td>
<td>v3.1</td>
<td>456</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Convert with round to nearest From Signed Word to floating-point format VX-form</td>
</tr>
<tr>
<td>vcfux</td>
<td>v2.03</td>
<td>425</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Convert with round to nearest From Signed Word to floating-point format VX-form</td>
</tr>
<tr>
<td>vcipher</td>
<td>v2.07</td>
<td>435</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector AES Cipher VX-form</td>
</tr>
<tr>
<td>vcipherlast</td>
<td>v2.07</td>
<td>435</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector AES Cipher Last VX-form</td>
</tr>
<tr>
<td>vclrb</td>
<td>v3.1</td>
<td>476</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Clear Leftmost Bytes VX-form</td>
</tr>
<tr>
<td>vclrb</td>
<td>v3.1</td>
<td>476</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Clear Rightmost Bytes VX-form</td>
</tr>
<tr>
<td>vclzd</td>
<td>v2.07</td>
<td>447</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Count Leading Zeros Byte VX-form</td>
</tr>
<tr>
<td>vclzd</td>
<td>v2.07</td>
<td>449</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Count Leading Zeros Doubleword VX-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 10 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vclzsm</td>
<td>v3.1</td>
<td>v3.1</td>
<td>v3.1</td>
<td>v3.1</td>
<td>v3.1</td>
<td>v3.1</td>
<td>v3.1</td>
<td>v3.1</td>
<td>v3.1</td>
<td>v3.1</td>
</tr>
<tr>
<td>vclzh</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
</tr>
<tr>
<td>vclzsb</td>
<td>v3.0</td>
<td>v3.0</td>
<td>v3.0</td>
<td>v3.0</td>
<td>v3.0</td>
<td>v3.0</td>
<td>v3.0</td>
<td>v3.0</td>
<td>v3.0</td>
<td>v3.0</td>
</tr>
<tr>
<td>vclzw</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
<td>v2.07</td>
</tr>
<tr>
<td>vcmpbf</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
</tr>
<tr>
<td>vcmpbfh</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
</tr>
<tr>
<td>vcmpbfj</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
</tr>
<tr>
<td>vcmpbfk</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
</tr>
<tr>
<td>vcmpbfq</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
<td>v2.03</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 11 of 30)

Appendix G. Power ISA Instruction Set Sorted by OpenPOWER Complian--- 1459
<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Book</th>
<th>Compliancy Subsets 2</th>
<th>Linux Optional Category 3</th>
<th>Always Optional Category 4</th>
<th>Mnemonic</th>
<th>Version 5</th>
<th>Privilege 6</th>
<th>Mode Dep 7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vctzw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>451</td>
<td>Vector Count Trailing Zeros Word VX-form</td>
</tr>
<tr>
<td>vdiveqsq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>361</td>
<td>Vector Divide Extended Signed Quadword VX-form</td>
</tr>
<tr>
<td>vdiveqsw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>363</td>
<td>Vector Divide Extended Signed Quadword VX-form</td>
</tr>
<tr>
<td>vdivequ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>359</td>
<td>Vector Divide Extended Unsigned Quadword VX-form</td>
</tr>
<tr>
<td>vdiveq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>361</td>
<td>Vector Divide Extended Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td>vdiveuq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>363</td>
<td>Vector Divide Extended Unsigned Quadword VX-form</td>
</tr>
<tr>
<td>vdiveu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>359</td>
<td>Vector Divide Extended Unsigned Word VX-form</td>
</tr>
<tr>
<td>vdiveu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>360</td>
<td>Vector Divide Signed Doubleword VX-form</td>
</tr>
<tr>
<td>vdiveq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>362</td>
<td>Vector Divide Signed Quadword VX-form</td>
</tr>
<tr>
<td>vdiveq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>358</td>
<td>Vector Divide Signed Word VX-form</td>
</tr>
<tr>
<td>vdiveuq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>360</td>
<td>Vector Divide Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td>vdiveu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>362</td>
<td>Vector Divide Unsigned Quadword VX-form</td>
</tr>
<tr>
<td>vdiveu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>358</td>
<td>Vector Divide Unsigned Word VX-form</td>
</tr>
<tr>
<td>vexpb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical Equivalence VX-form</td>
</tr>
<tr>
<td>vexpb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>466</td>
<td>Vector Expand Byte Mask VX-form</td>
</tr>
<tr>
<td>vexpb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>467</td>
<td>Vector Expand Doubleword Mask VX-form</td>
</tr>
<tr>
<td>vexpb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>466</td>
<td>Vector Expand Halfword Mask VX-form</td>
</tr>
<tr>
<td>vexpb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>468</td>
<td>Vector Expand Quadword Mask VX-form</td>
</tr>
<tr>
<td>vexpb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>467</td>
<td>Vector Expand Word Mask VX-form</td>
</tr>
<tr>
<td>vexpb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>432</td>
<td>Vector 2 Raised to the Exponent Estimate Floating-Point VX-form</td>
</tr>
<tr>
<td>vextdvvx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>312</td>
<td>Vector Extract Double Doubleword to VSR using GPR-specified Right-Index VA-form</td>
</tr>
<tr>
<td>vextdvvx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>312</td>
<td>Vector Extract Double Doubleword to VSR using GPR-specified Right-Index VA-form</td>
</tr>
<tr>
<td>vextdubvx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>309</td>
<td>Vector Extract Double Unsigned Byte to VSR using GPR-specified Left-Index VA-form</td>
</tr>
<tr>
<td>vextdubvx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>309</td>
<td>Vector Extract Double Unsigned Byte to VSR using GPR-specified Left-Index VA-form</td>
</tr>
<tr>
<td>vextdubvrx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>310</td>
<td>Vector Extract Double Unsigned Halfword to VSR using GPR-specified Left-Index VA-form</td>
</tr>
<tr>
<td>vextdubvrx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>310</td>
<td>Vector Extract Double Unsigned Halfword to VSR using GPR-specified Left-Index VA-form</td>
</tr>
<tr>
<td>vextdubh vx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>311</td>
<td>Vector Extract Double Signed Word to VSR using GPR-specified Left-Index VA-form</td>
</tr>
<tr>
<td>vextdubh vx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>311</td>
<td>Vector Extract Double Signed Word to VSR using GPR-specified Left-Index VA-form</td>
</tr>
<tr>
<td>vextdubh vrx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>311</td>
<td>Vector Extract Double Signed Word to VSR using GPR-specified Right-Index VA-form</td>
</tr>
<tr>
<td>vextdubh vrx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>311</td>
<td>Vector Extract Double Signed Word to VSR using GPR-specified Right-Index VA-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>471</td>
<td>Vector Extract Byte Mask VX-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>471</td>
<td>Vector Extract Doubleword Mask VX-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>471</td>
<td>Vector Extract Doubleword Mask VX-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>473</td>
<td>Vector Extract Quardword Mask VX-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>473</td>
<td>Vector Extract Quardword Mask VX-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>Vector Extract Unsigned Byte to VSR using immediate-specified index VX-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>Vector Extract Unsigned Halfword to VSR using immediate-specified index VX-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>305</td>
<td>Vector Extract Unsigned Word to VSR using immediate-specified index VX-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>372</td>
<td>Vector Extract Sign Byte To Doubleword VX-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>372</td>
<td>Vector Extract Sign Byte To Doubleword VX-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>372</td>
<td>Vector Extract Sign Byte To Doubleword VX-form</td>
</tr>
<tr>
<td>vextdibm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>372</td>
<td>Vector Extract Sign Byte To Doubleword VX-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 12 of 30)
<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Book</th>
<th>Compliancy Subset 2</th>
<th>Linux Optional Category 3</th>
<th>Always Optional Category 4</th>
<th>Mnemonic</th>
<th>Version 5</th>
<th>Privilege 6</th>
<th>Mode Dep 7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vxextsw2d</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>372</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vxextubix</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>306</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vxextubrix</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>306</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vxextuhxx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>307</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vxextuhxx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>307</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vxextuwxx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>308</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vxextuwxx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>308</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vgbbd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>445</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vgbbd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>446</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsblx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>315</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsblx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>315</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsbvx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>320</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsbvx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>320</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>319</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsdx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>318</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsdx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>318</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinserth</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>313</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinserth</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>313</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinserth</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>313</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsew</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>314</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsew</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>314</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsew</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>314</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsew</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>314</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>vinsew</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>314</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>viogelp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>433</td>
</tr>
<tr>
<td>000100</td>
<td>I...</td>
<td>I...</td>
<td>I...</td>
<td></td>
<td>viogelp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>423</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 13 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmaddp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Maximum Floating-Point VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmaddb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Maximum Signed Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmadds</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Maximum Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmaddsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Maximum Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmaddx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Maximum Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmaddub</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Maximum Unsigned Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmaddad</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Maximum Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmadduh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Maximum Unsigned Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmadduw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Maximum Unsigned Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmhraddhs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-High-Add Signed Halfword Saturate VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmhraddhs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vminfp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Minimum Floating-Point VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vminfb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Minimum Signed Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vminfd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Minimum Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vminfs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Minimum Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vminsw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Minimum Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vminub</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Minimum Unsigned Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vminud</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Minimum Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vminul</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Minimum Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vminuw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Minimum Unsigned Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmadduhm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmodsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Modulo Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmodsq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Modulo Signed Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmodsw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Modulo Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmodud</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Modulo Signed Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmoduh</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Modulo Signed Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmoduw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Modulo Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmrgew</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Merge Even Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmrgbh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Merge High Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmrghh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Merge High Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmrghw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Merge High Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmrglb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Merge Low Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmrglh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmrglw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Merge Low Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmrgow</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Merge Odd Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmsumcout</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-Sum &amp; write Carry-out Unsigned Doubleword VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmsummbm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-Sum Mixed Byte MODAL VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmsumshm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-Sum Signed Halfword MODAL VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmsums</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-Sum Signed Halfword Saturate VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmsumubm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-Sum Signed Mixed Byte MODAL VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmsumudm</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-Sum Signed Unaligned Doubleword MODAL VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmsumuhm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-Sum Signed Halfword MODAL VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmsums</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-Sum Signed Halfword Saturate VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10cuq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-by-10 Signed Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10cuq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Multiply-by-10 Extended &amp; write Carry-out Signed Quadword VX-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 14 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10equ</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>486</td>
<td>Vector Multiply-by-10 Extended Unsigned Quadword VX-form</td>
</tr>
<tr>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10eq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>487</td>
<td>Vector Multiply-by-10 Unsigned Quadword VX-form</td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10esb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>339</td>
<td>Vector Multiply Even Signed Byte VX-form</td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10esl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>346</td>
<td>Vector Multiply Even Signed Doubleword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10esh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>341</td>
<td>Vector Multiply Even Signed Halfword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10esw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>343</td>
<td>Vector Multiply Even Signed Word VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10ese</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>340</td>
<td>Vector Multiply Even Signed Byt VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10esd</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>342</td>
<td>Vector Multiply Even Signed Halfword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10esu</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>344</td>
<td>Vector Multiply Even Signed Word VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10hsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>349</td>
<td>Vector Multiply High Signed Doubleword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10hsw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>347</td>
<td>Vector Multiply High Signed Word VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10hse</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>349</td>
<td>Vector Multiply High Signed Doubleword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10hsw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>348</td>
<td>Vector Multiply High Signed Word VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10hlw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>350</td>
<td>Vector Multiply Low Doubleword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10osb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>339</td>
<td>Vector Multiply Odd Signed Byte VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10osd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>346</td>
<td>Vector Multiply Odd Signed Doubleword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10osh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>341</td>
<td>Vector Multiply Odd Signed Halfword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10osw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>343</td>
<td>Vector Multiply Odd Signed Word VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10oub</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>340</td>
<td>Vector Multiply Odd Unsigned Byte VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10oud</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>345</td>
<td>Vector Multiply Odd Unsigned Doubleword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10ouh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>342</td>
<td>Vector Multiply Odd Unsigned Halfword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10ouw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>344</td>
<td>Vector Multiply Odd Unsigned Word VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmul10oum</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>347</td>
<td>Vector Multiply Odd Unsigned Word Module VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnand</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical NAND VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vncipher</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>436</td>
<td>Vector AES Inverse Cipher VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnCipherLast</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>436</td>
<td>Vector AES Inverse Cipher Last VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnegd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>371</td>
<td>Vector Negate Doubleword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnegw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>371</td>
<td>Vector Negate Word VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnmsubfp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>423</td>
<td>Vector Negative Multiply-Subtract Floating-Point VA-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnor</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical NOR VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vorg</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical OR VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vorc</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical OR with Complement VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpdepd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>454</td>
<td>Vector Parallel Bits Deposit Doubleword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vperm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>296</td>
<td>Vector Permute VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpermr</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>296</td>
<td>Vector Permute Right-indexed VA-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpermxor</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>444</td>
<td>Vector Permute &amp; Exclusive-OR VA-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpslyt</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>455</td>
<td>Vector Parallel Bits Extract Doubleword VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpkpx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>278</td>
<td>Vector Pack Pixel VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpkdsds</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>281</td>
<td>Vector Pack Signed Doubleword Signed Saturate VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpkdsdu</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>281</td>
<td>Vector Pack Signed Doubleword Unsaturated VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpkshdu</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>279</td>
<td>Vector Pack Signed Halfword Unsaturated VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpkshus</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>279</td>
<td>Vector Pack Signed Halfword Unsaturated VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpswss</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>280</td>
<td>Vector Pack Signed Word Signed Saturate VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpswus</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>280</td>
<td>Vector Pack Signed Word Unsaturated VX-form</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpkudum</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>284</td>
<td>Vector Pack Signed Doubleword Unsized Module VX-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 15 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpkudus</td>
<td></td>
<td></td>
<td></td>
<td>284</td>
<td>Vector Pack Unsigned Doubleword Unsigned Saturate VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpkuhm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>282</td>
<td>Vector Pack Unsigned Halfword Unsigned Modulo VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpkuhus</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>282</td>
<td>Vector Pack Unsigned Halfword Unsigned Saturate VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpuwuum</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>283</td>
<td>Vector Pack Unsigned Word Unsigned Modulo VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpuwus</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>283</td>
<td>Vector Pack Unsigned Word Unsigned Saturate VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpmsumb</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>440</td>
<td>Vector Polynomial Multiply-Sum Byte VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpmsumd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>443</td>
<td>Vector Polynomial Multiply-Sum Doubleword VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpmsumv</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>441</td>
<td>Vector Polynomial Multiply-Sum Halfword VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpmsumw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>442</td>
<td>Vector Polynomial Multiply-Sum Word VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpopcntb</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>457</td>
<td>Vector Population Count Byte VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpopcnd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>458</td>
<td>Vector Population Count Doubleword VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpopcnth</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>457</td>
<td>Vector Population Count Halfword VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vpopcntw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>458</td>
<td>Vector Population Count Word VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vptvybd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>459</td>
<td>Vector Parity Byte Doubleword VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vptvybq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>460</td>
<td>Vector Parity Byte Quadword VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vptvysw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>459</td>
<td>Vector Parity Byte Word VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrefp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>434</td>
<td>Vector Reciprocal Estimate Floating-Point VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrfim</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>427</td>
<td>Vector Round to Floating-Point Integer toward -Infinity VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrfim</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>427</td>
<td>Vector Round to Floating-Point Integer Nearest VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrfip</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>428</td>
<td>Vector Round to Floating-Point Integer toward +Infinity VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrfiz</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>428</td>
<td>Vector Round to Floating-Point Integer toward Zero VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrlb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>404</td>
<td>Vector Rotate Left Byte VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrid</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>405</td>
<td>Vector Rotate Left Doubleword VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrdmi</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>401</td>
<td>Vector Rotate Left Doubleword then Mask Insert VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrdm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>408</td>
<td>Vector Rotate Left Doubleword then AND and Mask VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrlh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>404</td>
<td>Vector Rotate Left Halfword VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrlq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>405</td>
<td>Vector Rotate Left Quadradowd VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrlqm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>412</td>
<td>Vector Rotate Left Quadword then Mask Insert VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrlqmi</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>409</td>
<td>Vector Rotate Left Quadword then AND and Mask VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrlw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>404</td>
<td>Vector Rotate Left Word VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrlwi</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>410</td>
<td>Vector Rotate Left Word then Mask Insert VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vrlwmi</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>407</td>
<td>Vector Rotate Left Word then AND and Mask VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vsqrtelp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>434</td>
<td>Vector Reciprocals Square Root Estimate Floating-Point VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vsbbox</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>437</td>
<td>Vector AES SubBytes VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vsel</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>297</td>
<td>Vector Select VA-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vshasigmad</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>438</td>
<td>Vector SHA-512 Sigma Doubleword VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vshasigmas</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>439</td>
<td>Vector SHA-256 Sigma Word VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>val</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>300</td>
<td>Vector Shift Left VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vselb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>413</td>
<td>Vector Shift Left Byte VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>valid</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>414</td>
<td>Vector Shift Left Doubleword VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vsdbi</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>296</td>
<td>Vector Shift Left Double by Bit Immediate VN-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vsdpl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>298</td>
<td>Vector Shift Left Double by Octet Immediate VX-form</td>
</tr>
<tr>
<td>0011h</td>
<td>0011</td>
<td>001110</td>
<td></td>
<td></td>
<td>vsdh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>413</td>
<td>Vector Shift Left Halfword VX-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 16 of 30)
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Book</th>
<th>Compliancy Subsets²</th>
<th>Linux Optional Category³</th>
<th>Always Optional Category⁴</th>
<th>Mnemonic</th>
<th>Version⁵</th>
<th>Privilege⁶</th>
<th>Mode Dep⁷</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxsl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>301</td>
<td>Vector Shift Left by Octet VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxslq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>415</td>
<td>Vector Shift Left Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxsv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>302</td>
<td>Vector Shift Left Variable VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsvw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>414</td>
<td>Vector Shift Left Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vspltb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>293</td>
<td>Vector Splat Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vspltlh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>293</td>
<td>Vector Splat Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vspltlsb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>295</td>
<td>Vector Splat Immediate Signed Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vspltlsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>295</td>
<td>Vector Splat Immediate Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vspltlsaw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>295</td>
<td>Vector Splat Immediate Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vspltlw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>294</td>
<td>Vector Splat Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsr</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>300</td>
<td>Vector Shift Right VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrab</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>419</td>
<td>Vector Shift Right Algebraic Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrad</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>420</td>
<td>Vector Shift Right Algebraic Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrah</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>419</td>
<td>Vector Shift Right Algebraic Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsraq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>421</td>
<td>Vector Shift Right Algebraic Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsraw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>420</td>
<td>Vector Shift Right Algebraic Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>416</td>
<td>Vector Shift Right Byte VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>417</td>
<td>Vector Shift Right Doubleword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrdbi</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>299</td>
<td>Vector Shift Right Double by Bit Immediate VN-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsrh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>416</td>
<td>Vector Shift Right Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsro</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>301</td>
<td>Vector Shift Right by Octet VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>var</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>416</td>
<td>Vector Shift Right Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>varv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>302</td>
<td>Vector Shift Right Variable VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>varw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>417</td>
<td>Vector Shift Right Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsnh[rj]</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>474</td>
<td>Vector String Isolate Byte Left-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsnh[rj]</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>475</td>
<td>Vector String Isolate Byte Right-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsnh[rj]</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>476</td>
<td>Vector String Isolate Halfword Left-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsnh[rj]</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>474</td>
<td>Vector String Isolate Halfword Right-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubcuq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>338</td>
<td>Vector Subtract &amp; write Carry-out Unsigned Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubcuw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>331</td>
<td>Vector Subtract &amp; Write Carry-out Unsigned Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsbecuq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>338</td>
<td>Vector Subtract Extended &amp; write Carry-out Unsigned Quadword VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsbeuqnm</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>337</td>
<td>Vector Subtract Extended Unsigned Quadword Modulo VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubdp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>422</td>
<td>Vector Subtract Floating-Point VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubsbs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>331</td>
<td>Vector Subtract Signed Byte Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubsbs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>332</td>
<td>Vector Subtract Signed Word Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubsaws</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>332</td>
<td>Vector Subtract Signed Word Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsububm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>333</td>
<td>Vector Subtract Unsigned Byte Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsububs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>335</td>
<td>Vector Subtract Unsigned Byte Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubudm</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>334</td>
<td>Vector Subtract Unsigned Doubleword Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubuhm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>333</td>
<td>Vector Subtract Unaligned Halfword Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubahs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>335</td>
<td>Vector Subtract Unaligned Halfword Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubuqm</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>337</td>
<td>Vector Subtract Unaligned Module Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubuwem</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>334</td>
<td>Vector Subtract Unaligned Word Module VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubuws</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>336</td>
<td>Vector Subtract Unaligned Word Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsum2sws</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>368</td>
<td>Vector Sum across Half Signed Word Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsum4sbs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>369</td>
<td>Vector Sum across Quarter Signed Byte Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsum4shs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>369</td>
<td>Vector Sum across Quarter Signed Halfword Saturate VX-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 17 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>001100 ...... 001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsum4ubs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>370</td>
<td>Vector Sum across Quarter Unsigned Byte Saturate VX-form</td>
</tr>
<tr>
<td>001100 ...... 001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsmsws</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>367</td>
<td>Vector Sum across Signed Word Saturate VX-form</td>
</tr>
<tr>
<td>001100 ...... 001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupkhpix</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Unpack High Pixel VX-form</td>
</tr>
<tr>
<td>001100 ...... 001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupkhsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Unpack High Signed Halfword VX-form</td>
</tr>
<tr>
<td>001100 ...... 001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupkhs</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>287</td>
<td>Vector Unpack High Signed Word VX-form</td>
</tr>
<tr>
<td>001100 ...... 001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupkltlpx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Unpack Low Pixel VX-form</td>
</tr>
<tr>
<td>001100 ...... 001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupklsb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Unpack Low Signed Byte VX-form</td>
</tr>
<tr>
<td>001100 ...... 001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupklhs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Unpack Low Signed Halfword VX-form</td>
</tr>
<tr>
<td>001100 ...... 001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vupkls</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>287</td>
<td>Vector Unpack Low Signed Word VX-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxor</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical XOR VX-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxabsdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>658</td>
<td>VSX Scalar Absolute Double-Precision XX2-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxaddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>659</td>
<td>VSX Scalar Add Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxadsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>664</td>
<td>VSX Scalar Add Single-Precision XX3-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>670</td>
<td>VSX Scalar Compare Equal Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>671</td>
<td>VSX Scalar Compare Equal Quad-Precision X-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>668</td>
<td>VSX Scalar Compare Exponents Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>672</td>
<td>VSX Scalar Compare Greater Than Equal Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>673</td>
<td>VSX Scalar Compare Greater Than Equal Quad-Precision X-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>674</td>
<td>VSX Scalar Compare Greater Than Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>675</td>
<td>VSX Scalar Compare Greater Than Quad-Precision X-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>676</td>
<td>VSX Scalar Compare Ordered Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>679</td>
<td>VSX Scalar Compare Unordered Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>682</td>
<td>VSX Scalar Copy Sign Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>683</td>
<td>VSX Scalar Convert with round Double-Precision to Half-Precision format XX2-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>685</td>
<td>VSX Scalar Convert with round Double-Precision to Single-Precision format XX2-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>688</td>
<td>VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling IQ2-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>687</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format XX2-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>689</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Word format XX2-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>691</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>693</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format XX2-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>695</td>
<td>VSX Scalar Convert Half-Precision to Double-Precision format XX2-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>699</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Quadword X-form</td>
</tr>
<tr>
<td>111100 ...... 001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcompqdpdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>705</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Unsigned Quadword X-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 18 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcqsdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>709</td>
<td>VXS Scalar Convert Single-Precision to Double-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcqvdp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>710</td>
<td>VXS Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvsqdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>711</td>
<td>VXS Scalar Convert with round Signed Quadword to Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvsxdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>712</td>
<td>VXS Scalar Convert with round Signed Doubleword to Double-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvsxdsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>713</td>
<td>VXS Scalar Convert with round Signed Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcsvnqsp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>714</td>
<td>VXS Scalar Convert with round Unsigned Quadword to Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvsxaddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>715</td>
<td>VXS Scalar Convert with round Unsigned Doubleword to Double-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcsvxaddp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>716</td>
<td>VXS Scalar Convert with round Unsigned Doubleword to Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xdivdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>717</td>
<td>VXS Scalar Divide Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xdivsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>718</td>
<td>VXS Scalar Divide Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xalexspdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>725</td>
<td>VXS Scalar Insert Exponent Double-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaddadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>725</td>
<td>VXS Scalar Multiply-Add Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaddasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>728</td>
<td>VXS Scalar Multiply-Add Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaddmsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>728</td>
<td>VXS Scalar Multiply-Add Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaddmsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>728</td>
<td>VXS Scalar Multiply-Add Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmacdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>736</td>
<td>VXS Scalar Maximum Type-C Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaccdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>736</td>
<td>VXS Scalar Maximum Type-C Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>734</td>
<td>VXS Scalar Maximum Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmajdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>739</td>
<td>VXS Scalar Maximum Type-J Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmencd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>743</td>
<td>VXS Scalar Minimum Type-C Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmencdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>745</td>
<td>VXS Scalar Minimum Type-C Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmindp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>741</td>
<td>VXS Scalar Minimum Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmnindp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>746</td>
<td>VXS Scalar Minimum Type-J Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmsubdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>748</td>
<td>VXS Scalar Multiply-Subtract Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmsubasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>751</td>
<td>VXS Scalar Multiply-Subtract Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmsubmsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>751</td>
<td>VXS Scalar Multiply-Subtract Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmudp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>757</td>
<td>VXS Scalar Multiply Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmulp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>761</td>
<td>VXS Scalar Multiply Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnabsdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>763</td>
<td>VXS Scalar Absolute Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnegdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>764</td>
<td>VXS Scalar Negate Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmaddap</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>765</td>
<td>VXS Scalar Negative Multiply-Add Type-A Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmaddsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>770</td>
<td>VXS Scalar Negative Multiply-Add Type-A Single-Precision XX3-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 19 of 30)
<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Book</th>
<th>Compliancy Subsets 2</th>
<th>Linux Optional Category 3</th>
<th>Always Optional Category 4</th>
<th>Mnemonic</th>
<th>Version 5</th>
<th>Privilege 6</th>
<th>Mode Dep 7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnaddmdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>765</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>770</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>776</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsnmsubasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>779</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>776</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>777</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>785</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>786</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>787</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>788</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>789</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>790</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>791</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>797</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>798</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>799</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>801</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>805</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>807</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>811</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>813</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>814</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>815</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>817</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>818</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>819</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>820</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>821</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>825</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>832</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>833</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>834</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>835</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 20 of 30)
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Book</th>
<th>Compilator Subsets²</th>
<th>Linux Optional Category³</th>
<th>Always Optional Category⁴</th>
<th>Mnemonic</th>
<th>Version⁵</th>
<th>Privilege⁶</th>
<th>Mode Dep⁷</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcmpgdp[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>836</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcmpgdp[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>837</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcpsgndp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>838</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcpsgnsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>838</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvbf16sp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>839</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvdpssp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>840</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvdpaxds</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>841</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvdpaxws</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>843</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvdpaxds</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>845</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvdpuxws</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>847</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvhpssp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>849</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvspsbf16</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>850</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvspsdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>851</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvspsdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>852</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvspsdxs</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>853</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvspsdxs</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>855</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvpsdxs</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>857</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvpsdxs</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>859</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvpsdxs</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>861</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvssdx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>862</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvssdx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>863</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvssxsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>864</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvuxdx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>865</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvuxdx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>866</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvuxdx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>866</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvuxpsp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>867</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvdvdp[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>868</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvdvpsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>869</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xveespdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>869</td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xveespdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>869</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 21 of 30)
<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Book</th>
<th>Compliancy Subsets 2</th>
<th>Linux Optional Category 3</th>
<th>Always Optional Category 4</th>
<th>Mnemonic</th>
<th>Version 5</th>
<th>Privilege 6</th>
<th>Mode Dep 7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xuaddadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>897</td>
<td>VSX Vector Multiply-Add Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xumaddsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>900</td>
<td>VSX Vector Multiply-Add Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xumaddmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>897</td>
<td>VSX Vector Multiply-Add Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xumaddmsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>900</td>
<td>VSX Vector Multiply-Add Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xumaxdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>903</td>
<td>VSX Vector Maximum Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xumaxsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>905</td>
<td>VSX Vector Maximum Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmindp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>907</td>
<td>VSX Vector Minimum Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvminsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>909</td>
<td>VSX Vector Minimum Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmsubadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>911</td>
<td>VSX Vector Multiply-Subtract Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmsubasp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>914</td>
<td>VSX Vector Multiply-Subtract Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmsubmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>911</td>
<td>VSX Vector Multiply-Subtract Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmultsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>917</td>
<td>VSX Vector Multiply Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmultsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>919</td>
<td>VSX Vector Multiply Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmabsdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>921</td>
<td>VSX Vector Negative Absolute Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvnegdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>922</td>
<td>VSX Vector Negative Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvnegsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>922</td>
<td>VSX Vector Negative Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmaddadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>923</td>
<td>VSX Vector Negative Multiply-Add Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmaddasp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>927</td>
<td>VSX Vector Negative Multiply-Add Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmaddmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>923</td>
<td>VSX Vector Negative Multiply-Add Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmaddmsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>927</td>
<td>VSX Vector Negative Multiply-Add Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmsubadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>930</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmsubasp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>933</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmsubbmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>930</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvmsubbmsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>933</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>936</td>
<td>VSX Vector Round to Double-Precision Integer using round to Nearest Away XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvdpic</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>937</td>
<td>VSX Vector Round to Double-Precision Integer Exact using Current rounding mode XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvdpim</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>938</td>
<td>VSX Vector Round to Double-Precision Integer using round toward -Infinity XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvdpip</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>939</td>
<td>VSX Vector Round to Double-Precision Integer using round toward +Infinity XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvdpiz</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>939</td>
<td>VSX Vector Round to Double-Precision Integer using round toward Zero XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvredp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>940</td>
<td>VSX Vector Reciprocal Estimate Double-Precision XX2-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 22 of 30)
Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 23 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subset</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxgencdvm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>976</td>
<td>VSX Vector Generate PCV from Doubleword Mask X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxgencdfm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>972</td>
<td>VSX Vector Generate PCV from Halfword Mask X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxgenpawkm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>974</td>
<td>VSX Vector Generate PCV from Word Mask X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxgenpawm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>969</td>
<td>VSX Vector Insert Word XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxlandw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>978</td>
<td>VSX Vector Logical AND XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxlandc</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>978</td>
<td>VSX Vector Logical AND with Complement XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxleqv</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>979</td>
<td>VSX Vector Logical Equivalence XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxlhrand</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>979</td>
<td>VSX Vector Logical NAND XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxlnor</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>980</td>
<td>VSX Vector Logical NOR XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxldor</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>981</td>
<td>VSX Vector Logical OR XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxlorc</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>980</td>
<td>VSX Vector Logical OR with Complement XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxlorx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>981</td>
<td>VSX Vector Logical XOR XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxmrghw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>982</td>
<td>VSX Vector Merge High Word XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxmrhlw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>982</td>
<td>VSX Vector Merge Low Word XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxperm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>985</td>
<td>VSX Vector Permute XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxpermdd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>986</td>
<td>VSX Vector Permute Doubleword Immediate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxpermrr</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>985</td>
<td>VSX Vector Permute Right-indexed XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxpermrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>987</td>
<td>VSX Vector Permute Extended 8RR:XX4-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxssel</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>988</td>
<td>VSX Vector Select XX4-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsidwi</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>990</td>
<td>VSX Vector Shift Left Double by Word Immediate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsplitl32dx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>992</td>
<td>VSX Vector Split Immediate32 Doubleword Indexed 8RR:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsplitlbb</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>991</td>
<td>VSX Vector Split Immediate Byte X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsplitlfp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>991</td>
<td>VSX Vector Split Immediate Double-Precision 8RR:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsplitlw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>992</td>
<td>VSX Vector Split Immediate Word 8RR:D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsplitlw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>993</td>
<td>VSX Vector Split Word XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>copy</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>1068</td>
<td>Copy X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cpabort</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>1069</td>
<td>Copy-Paste Abort X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>darn</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>86</td>
<td>Deliver A Random Number X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcbf</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1064</td>
<td>Data Cache Block Flush X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcbst</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1063</td>
<td>Data Cache Block Store X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dbzb</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1061</td>
<td>Data Cache Block Touch X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dbzbst</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1062</td>
<td>Data Cache Block Touch for Store X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dbz</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1063</td>
<td>Data Cache Block set to Zero X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dieceo</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1088</td>
<td>Enforce In-order Execution of I/O X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>hrfd</td>
<td>v2.02</td>
<td>HV</td>
<td></td>
<td>1152</td>
<td>Return From Interrupt Doubleword Hypervisor XL-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>idz</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1052</td>
<td>Instruction Cache Block Invalidate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>icbt</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>1052</td>
<td>Instruction Cache Block Touch X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>isel</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>98</td>
<td>Integer Select X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sync</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1076</td>
<td>Instruction Synchronize XL-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lbarx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>1077</td>
<td>Load Byte And Reserve Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lbzcx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1164</td>
<td>Load Byte &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ldx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1164</td>
<td>Load Doubleword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fdp</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>159</td>
<td>Load Floating-Point Double Pair DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fdpx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>159</td>
<td>Load Floating-Point Double Pair Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fharx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>1078</td>
<td>Load Halfword And Reserve Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fhrx</td>
<td>PPC</td>
<td></td>
<td></td>
<td>67</td>
<td>Load Halfword Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fhzcx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1164</td>
<td>Load Halfword &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 24 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwzci</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1164</td>
<td>Load Word &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msgcr</td>
<td>v2.07</td>
<td>HV</td>
<td></td>
<td>1329</td>
<td>Message Clear X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msgcrdp</td>
<td>v2.07</td>
<td>P</td>
<td></td>
<td>1331</td>
<td>Message Clear Privileged X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msgcmd</td>
<td>v2.07</td>
<td>HV</td>
<td></td>
<td>1326</td>
<td>Message Send X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msgcmdp</td>
<td>v2.07</td>
<td>P</td>
<td></td>
<td>1330</td>
<td>Message Send Privileged X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msgsync</td>
<td>v3.0</td>
<td>HV</td>
<td></td>
<td>1331</td>
<td>Message Synchronize X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mltmd</td>
<td>PPC</td>
<td>P</td>
<td>SR</td>
<td>1175</td>
<td>Move To MSR Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbie</td>
<td>PPC</td>
<td>P</td>
<td>SR</td>
<td>1224</td>
<td>SLB Invalidate All X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbiag</td>
<td>v3.05</td>
<td>P</td>
<td></td>
<td>1226</td>
<td>SLB Invalidate All Global X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbie</td>
<td>PPC</td>
<td>P</td>
<td></td>
<td>1221</td>
<td>SLB Invalidate Entry X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbie</td>
<td>v3.0</td>
<td>P</td>
<td></td>
<td>1222</td>
<td>SLB Invalidate Entry Global X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbfree</td>
<td>v2.00</td>
<td>P</td>
<td></td>
<td>1229</td>
<td>SLB Move From Entry ESIO X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbfreev</td>
<td>v2.00</td>
<td>P</td>
<td></td>
<td>1228</td>
<td>SLB Move From Entry VSIO X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slbsync</td>
<td>v3.0</td>
<td>P</td>
<td></td>
<td>1227</td>
<td>SLB Move To Entry X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stbcix</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Byte Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stbcr</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>1079</td>
<td>Store Byte Conditional Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdcx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Doubleword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdcx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Halfword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdhcx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>1080</td>
<td>Store Halfword Conditional Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stmrw</td>
<td>P1</td>
<td></td>
<td></td>
<td>70</td>
<td>Store Multiple Word D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stop</td>
<td>v3.0</td>
<td>P</td>
<td></td>
<td>1155</td>
<td>Stop XL-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sq</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>66</td>
<td>Store Quadword DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stqcr</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Store Quadword Conditional Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stswx</td>
<td>P1</td>
<td></td>
<td></td>
<td>73</td>
<td>Store String Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stswx</td>
<td>P1</td>
<td></td>
<td></td>
<td>73</td>
<td>Store String Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stwox</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Word Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tblie</td>
<td>P1</td>
<td>HV</td>
<td>64</td>
<td>1231</td>
<td>TLB Invalidate Entry X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tblie</td>
<td>v2.03</td>
<td>P</td>
<td>64</td>
<td>1236</td>
<td>TLB Invalidate Entry Local X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tblsync</td>
<td>PPC</td>
<td>HV/IP</td>
<td>1240</td>
<td></td>
<td>TLB Synchronize X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>1090</td>
<td>Wait X-form</td>
</tr>
<tr>
<td></td>
<td>AMO</td>
<td></td>
<td></td>
<td></td>
<td>idat</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>1073</td>
<td>Load Doubleword Atomic X-form</td>
</tr>
<tr>
<td></td>
<td>AMO</td>
<td></td>
<td></td>
<td></td>
<td>kwat</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>1073</td>
<td>Load Word Atomic X-form</td>
</tr>
<tr>
<td></td>
<td>AMO</td>
<td></td>
<td></td>
<td></td>
<td>stdat</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>1075</td>
<td>Store Doubleword Atomic X-form</td>
</tr>
<tr>
<td></td>
<td>AMO</td>
<td></td>
<td></td>
<td></td>
<td>stwait</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>1075</td>
<td>Store Word Atomic X-form</td>
</tr>
<tr>
<td></td>
<td>BFP12</td>
<td></td>
<td></td>
<td></td>
<td>xsabog</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>656</td>
<td>VSX Scalar Absolute Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td>BFP12</td>
<td></td>
<td></td>
<td></td>
<td>xsaddq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>666</td>
<td>VSX Scalar Add Quad-Precision [using round to Odd] X-form</td>
</tr>
<tr>
<td></td>
<td>BFP12</td>
<td></td>
<td></td>
<td></td>
<td>xscmpeqq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>669</td>
<td>VSX Scalar Compare Exponents Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td>BFP12</td>
<td></td>
<td></td>
<td></td>
<td>xsncmpq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>678</td>
<td>VSX Scalar Compare Ordered Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td>BFP12</td>
<td></td>
<td></td>
<td></td>
<td>xsncmpuq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>681</td>
<td>VSX Scalar Compare Unordered Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td>BFP12</td>
<td></td>
<td></td>
<td></td>
<td>xscmpeq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>682</td>
<td>VSX Scalar Copy Sign Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td>BFP12</td>
<td></td>
<td></td>
<td></td>
<td>xsavd7</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>684</td>
<td>VSX Scalar Convert Double-Precision to Quad-Precision format X-form</td>
</tr>
<tr>
<td></td>
<td>BFP12</td>
<td></td>
<td></td>
<td></td>
<td>xsavdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>696</td>
<td>VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] X-form</td>
</tr>
<tr>
<td></td>
<td>BFP12</td>
<td></td>
<td></td>
<td></td>
<td>xsavdpsd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>697</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format X-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 25 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
<tr>
<td>111111 01001 111110110000 001001001</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 26 of 30)
## Appendix G. Power ISA Instruction Set Sorted by OpenPOWER Compliancy Subset

<table>
<thead>
<tr>
<th>Instruction&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Book</th>
<th>Compliancy Subset&lt;sup&gt;2&lt;/sup&gt;</th>
<th>Linux Optional Category&lt;sup&gt;3&lt;/sup&gt;</th>
<th>Always Optional Category&lt;sup&gt;4&lt;/sup&gt;</th>
<th>Mnemonic</th>
<th>Version&lt;sup&gt;5&lt;/sup&gt;</th>
<th>Privilege&lt;sup&gt;6&lt;/sup&gt;</th>
<th>Mode Disp&lt;sup&gt;7&lt;/sup&gt;</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>didiv</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>209</td>
<td>DFP Divide Quad X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>denvld</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>235</td>
<td>DFP Encode BCD To DPD X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>denvlq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>235</td>
<td>DFP Encode BCD To DPD Quad X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>denvlo</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Insert Based Exponent X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>denvll</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Insert Based Exponent Quad X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmuli</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>208</td>
<td>DFP Multiply X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmulq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>208</td>
<td>DFP Multiply Quad X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dqiul</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>219</td>
<td>DFP Quantize Z23-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dqul</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>217</td>
<td>DFP Quantize Immediate Z23-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dqulq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>217</td>
<td>DFP Quantize Immediate Quad Z23-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dqulq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>217</td>
<td>DFP Quantize Immediate Quad Z23-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dpdpq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>230</td>
<td>DFP Round To DFP Long X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dprintf</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>226</td>
<td>DFP Round To FP Integer Without Inexact Z23-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dprintln</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>226</td>
<td>DFP Round To FP Integer Without Inexact Quad Z23-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmisi</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>224</td>
<td>DFP Round To FP Integer With Inexact Z23-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmisq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>224</td>
<td>DFP Round To FP Integer With Inexact Quad Z23-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmsi</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>221</td>
<td>DFP Round Z23-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmsq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>221</td>
<td>DFP Round Quad Z23-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmsi</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>230</td>
<td>DFP Round To DFP Short X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmsq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Shift Significand Left Immediate Z22-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmsi</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Shift Significand Left Immediate Quad Z22-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmsq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Shift Significand Right Immediate Z22-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmsi</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Shift Significand Right Immediate Quad Z22-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmsi</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>206</td>
<td>DFP Subtract X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dmsq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>206</td>
<td>DFP Subtract Quad X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dstitc</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>213</td>
<td>DFP Test Data Class Z22-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dstitcq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>213</td>
<td>DFP Test Data Class Quad Z22-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dstitd</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>213</td>
<td>DFP Test Data Group Z22-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dstitdq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>213</td>
<td>DFP Test Data Group Quad Z22-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dstitex</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>214</td>
<td>DFP Test Exponent X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dstitux</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>214</td>
<td>DFP Test Exponent Quad X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dstitf</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>215</td>
<td>DFP Test Significance X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dstitlf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>216</td>
<td>DFP Test Significance Immediate X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dstitlq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>216</td>
<td>DFP Test Significance Immediate Quad X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>dstitlq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>215</td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>ditex</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Extract Based Exponent X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>ditexq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>236</td>
<td>DFP Extract Based Exponent Quad X-form</td>
</tr>
<tr>
<td>...</td>
<td>DPP</td>
<td></td>
<td></td>
<td></td>
<td>ddrbb</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>1153</td>
<td>Ultraviolet Return From Interrupt Doubleword XL-form</td>
</tr>
<tr>
<td>...</td>
<td>MMA</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) MMIRR:XX3-form</td>
<td></td>
<td></td>
<td>magintu</td>
<td>v3.0C UV</td>
<td></td>
<td></td>
<td>1327</td>
<td>Ultraviolet Message Clear X-form</td>
</tr>
<tr>
<td>...</td>
<td>MMA</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) MMIRR:XX3-form</td>
<td></td>
<td></td>
<td>magintv</td>
<td>v3.0C UV</td>
<td></td>
<td></td>
<td>1327</td>
<td>Ultraviolet Message SendX-form</td>
</tr>
<tr>
<td>...</td>
<td>MMA</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) MMIRR:XX3-form</td>
<td></td>
<td></td>
<td>magintw</td>
<td>v3.0C UV</td>
<td></td>
<td></td>
<td>1153</td>
<td>Ultraviolet Return From Interrupt Doubleword XL-form</td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 27 of 30)
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Book</th>
<th>Compliancy Subsets²</th>
<th>Linux Optional Category³</th>
<th>Always Optional Category⁴</th>
<th>Mnemonic</th>
<th>Version⁵</th>
<th>Privilege⁶</th>
<th>Mode Dep⁷</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf16ger2pp</td>
<td>v3.1</td>
<td>827</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf16ger2</td>
<td>v3.1</td>
<td>871</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf16ger2nn</td>
<td>v3.1</td>
<td>871</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf16ger2np</td>
<td>v3.1</td>
<td>871</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf16ger2pn</td>
<td>v3.1</td>
<td>871</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf32ger</td>
<td>v3.1</td>
<td>875</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf32germn</td>
<td>v3.1</td>
<td>875</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf32germp</td>
<td>v3.1</td>
<td>875</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf32gerp</td>
<td>v3.1</td>
<td>875</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf64ger</td>
<td>v3.1</td>
<td>879</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf64germn</td>
<td>v3.1</td>
<td>879</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf64germp</td>
<td>v3.1</td>
<td>879</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxvf64gerp</td>
<td>v3.1</td>
<td>879</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxv16ger2</td>
<td>v3.1</td>
<td>891</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxv16ger2pp</td>
<td>v3.1</td>
<td>891</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxv16ger2s</td>
<td>v3.1</td>
<td>893</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxv16ger2pp</td>
<td>v3.1</td>
<td>893</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxv4ger8</td>
<td>v3.1</td>
<td>883</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001 11100</td>
<td>I ...</td>
<td>MIA</td>
<td>MIA</td>
<td>pmxv4ger8pp</td>
<td>v3.1</td>
<td>883</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 28 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMA MMA xvi16ger4</td>
<td>MMA</td>
<td>MMA</td>
<td>pmxvi8ger4</td>
<td>v3.1</td>
<td>886</td>
<td>Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) MMIRR:XX3-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvi16ger4pp</td>
<td>MMA</td>
<td>MMA</td>
<td>pmxvi8ger4pp</td>
<td>v3.1</td>
<td>886</td>
<td>Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate MMIRR:XX3-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvi8ger4spp</td>
<td>MMA</td>
<td>MMA</td>
<td>pmxvi8ger4spp</td>
<td>v3.1</td>
<td>889</td>
<td>Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturate Positive multiply, Positive accumulate MMIRR:XX3-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf16ger2</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf16ger2</td>
<td>v3.1</td>
<td>827</td>
<td>VSX Vector bfloat16 GER (Rank-2 Update) X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf16ger2nm</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf16ger2nm</td>
<td>v3.1</td>
<td>827</td>
<td>VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Positive accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf16ger2np</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf16ger2np</td>
<td>v3.1</td>
<td>827</td>
<td>VSX Vector bfloat16 GER (Rank-2 Update) Negative multiply, Positive accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf16ger2nn</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf16ger2nn</td>
<td>v3.1</td>
<td>827</td>
<td>VSX Vector bfloat16 GER (Rank-2 Update) Positive multiply, Negative accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf16ger2</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf16ger2</td>
<td>v3.1</td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf16ger2n</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf16ger2n</td>
<td>v3.1</td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf16ger2p</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf16ger2p</td>
<td>v3.1</td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf16ger2pn</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf16ger2pn</td>
<td>v3.1</td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf16ger2pp</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf16ger2pp</td>
<td>v3.1</td>
<td>871</td>
<td>VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf32ger</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf32ger</td>
<td>v3.1</td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf32germ</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf32germ</td>
<td>v3.1</td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf32germp</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf32germp</td>
<td>v3.1</td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf32gerpn</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf32gerpn</td>
<td>v3.1</td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf32gerpp</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf32gerpp</td>
<td>v3.1</td>
<td>875</td>
<td>VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf64ger</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf64ger</td>
<td>v3.1</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf64germ</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf64germ</td>
<td>v3.1</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf64germp</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf64germp</td>
<td>v3.1</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf64gerpn</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf64gerpn</td>
<td>v3.1</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvf64gerpp</td>
<td>MMA</td>
<td>MMA</td>
<td>xvf64gerpp</td>
<td>v3.1</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvi16ger2</td>
<td>MMA</td>
<td>MMA</td>
<td>xvi16ger2</td>
<td>v3.1</td>
<td>891</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvi16ger2p</td>
<td>MMA</td>
<td>MMA</td>
<td>xvi16ger2p</td>
<td>v3.1</td>
<td>891</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate X00-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMA MMA xvi16ger2pp</td>
<td>MMA</td>
<td>MMA</td>
<td>xvi16ger2pp</td>
<td>v3.1</td>
<td>893</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturate X00-form</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Figure 93. Power ISA AS Instruction Set Sorted by Compliancy Subset (Sheet 30 of 30)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111011 ...</td>
<td>MMA</td>
<td>X..</td>
<td>.X.</td>
<td>..X.</td>
<td>xvi16ger2spp</td>
<td>v3.1</td>
<td>893</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111011 ...</td>
<td>MMA</td>
<td>X..</td>
<td>.X.</td>
<td>..X.</td>
<td>xvi4ger8</td>
<td>v3.1</td>
<td>883</td>
<td>VSX Vector 4-bit Signed Integer GER (rank-8 update) XX3-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111011 ...</td>
<td>MMA</td>
<td>X..</td>
<td>.X.</td>
<td>..X.</td>
<td>xvi4ger8pp</td>
<td>v3.1</td>
<td>883</td>
<td>VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111011 ...</td>
<td>MMA</td>
<td>X..</td>
<td>.X.</td>
<td>..X.</td>
<td>xvi8ger4</td>
<td>v3.1</td>
<td>886</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) XX3-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111011 ...</td>
<td>MMA</td>
<td>X..</td>
<td>.X.</td>
<td>..X.</td>
<td>xvi8ger4pp</td>
<td>v3.1</td>
<td>886</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111011 ...</td>
<td>MMA</td>
<td>X..</td>
<td>.X.</td>
<td>..X.</td>
<td>xvi8ger4spp</td>
<td>v3.1</td>
<td>889</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturate Positive multiply, Positive accumulate XX3-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111011 ...</td>
<td>MMA</td>
<td>X..</td>
<td>.X.</td>
<td>..X.</td>
<td>xmmfacec</td>
<td>v3.1</td>
<td>983</td>
<td>VSX Move From Accumulator X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111011 ...</td>
<td>MMA</td>
<td>X..</td>
<td>.X.</td>
<td>..X.</td>
<td>xmmfacec</td>
<td>v3.1</td>
<td>984</td>
<td>VSX Move To Accumulator X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111011 ...</td>
<td>MMA</td>
<td>X..</td>
<td>.X.</td>
<td>..X.</td>
<td>xmmsetaccz</td>
<td>v3.1</td>
<td>989</td>
<td>VSX Set Accumulator to Zero X-form</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 1. Instruction
- /: Instruction bit that corresponds to a reserved field, must have a value of 0, otherwise invalid form.
- #: Instruction bit that corresponds to an operand bit, may have a value of either 0 or 1.
- $: Instruction bit that corresponds to an opcode bit having a value 0.
- #: Instruction bit that corresponds to an opcode bit having a value 1.

### 2. OpenPOWER Compliancy Subsets
- X..: Instruction included in the Scalar Fixed-Point Compliancy subset
- .X.: Instruction included in the Scalar Fixed-Point + Floating-Point Compliancy subset.
- ..X.: Instruction included in the Linux Compliancy subset.
- ...X: Instruction included in the AIX Compliancy subset.

### 3. Linux Optional Category
- AMO: Instruction part of Atomic Memory Operations category.
- BFP128: Instruction part of Quad-Precision Floating-Point category.
- BHRB: Instruction part of Branch History Rolling Buffer category.
- DFP: Instruction part of Decimal Floating-Point category.
- EBB: Instruction part of Event-Based Branch category.
- MMA: Instruction part of Matrix-Multiplication Assist category.

### 4. Always Optional Category
- MMA: Instruction part of Matrix-Multiplication Assist category.
### 5. Version

- **P1**: Instruction introduced in POWER Architecture.
- **P2**: Instruction introduced in POWER2 Architecture.
- **PPC**: Instruction introduced in PowerPC Architecture prior to v2.00.
- **v2.00**: Instruction introduced in PowerPC Architecture Version 2.00.
- **v2.01**: Instruction introduced in PowerPC Architecture Version 2.01.
- **v2.02**: Instruction introduced in PowerPC Architecture Version 2.02.
- **v2.03**: Instruction introduced in Power ISA Version 2.03.
- **v2.04**: Instruction introduced in Power ISA Version 2.04.
- **v2.05**: Instruction introduced in Power ISA Version 2.05.
- **v2.06**: Instruction introduced in Power ISA Version 2.06.
- **v2.07**: Instruction introduced in Power ISA Version 2.07.
- **v3.0**: Instruction introduced in Power ISA Version 3.0.
- **v3.0B**: Instruction introduced in Power ISA Version 3.0B.
- **v3.0C**: Instruction introduced in Power ISA Version 3.0C.
- **v3.1**: Instruction introduced in Power ISA Version 3.1.

### 6. Privilege

- **P**: Denotes an instruction that is treated as privileged.
- **O**: Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor-privileged for mtspr), depending on the SPR or PMR number.
- **PI**: Denotes an instruction that is illegal in privileged state.
- **HV**: Denotes an instruction that can be executed only in hypervisor state.
- **UV**: Denotes an instruction that can be executed only in ultravisor state.

### 7. Mode Dependency

Except as described below and in Section 1.10.3, “Effective Address Calculation”, in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

- **CT**: If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
- **SR**: The setting of status registers (such as XER and CR0) is mode-dependent.
- **SF=1**: The instruction can be executed only in 64-bit mode.
Appendix H. Power ISA Instruction Set Sorted by Mnemonic

This appendix lists all the instructions in the Power ISA, sorted by mnemonic.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>add[j]</td>
<td>P1</td>
<td>SR</td>
<td>77</td>
<td></td>
<td>Add XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addc[j]</td>
<td>P1</td>
<td>SR</td>
<td>78</td>
<td></td>
<td>Add Carrying XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi[j]</td>
<td>P1</td>
<td>SR</td>
<td>78</td>
<td></td>
<td>Add Carrying &amp; record OV XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi0[j]</td>
<td>P1</td>
<td>SR</td>
<td>78</td>
<td></td>
<td>Add Extended XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi0[j]</td>
<td>P1</td>
<td>SR</td>
<td>78</td>
<td></td>
<td>Add Extended &amp; record OV XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addiex</td>
<td>v3.0B</td>
<td>80 68</td>
<td>Add Extended using alternate carry bit Z23-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi6s</td>
<td>v2.06</td>
<td>118</td>
<td>Add and Generate Sines XO-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>P1</td>
<td>76</td>
<td></td>
<td></td>
<td>Add Immediate D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi0</td>
<td>P1</td>
<td>SR</td>
<td>77</td>
<td></td>
<td>Add Immediate Carrying D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi0</td>
<td>P1</td>
<td>SR</td>
<td>77</td>
<td></td>
<td>Add Immediate Carrying and Record D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addis</td>
<td>P1</td>
<td>76</td>
<td></td>
<td></td>
<td>Add Immediate Shifted D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi0[j]</td>
<td>P1</td>
<td>SR</td>
<td>79</td>
<td></td>
<td>Add to Minus One Extended XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi0[j]</td>
<td>P1</td>
<td>SR</td>
<td>79</td>
<td></td>
<td>Add to Minus One Extended &amp; record OV XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi0[j]</td>
<td>P1</td>
<td>SR</td>
<td>77</td>
<td></td>
<td>Add &amp; record OV XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi0[j]</td>
<td>P1</td>
<td>76</td>
<td></td>
<td></td>
<td>Add PC Immediate Shifted DX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi0[j]</td>
<td>P1</td>
<td>SR</td>
<td>79</td>
<td></td>
<td>Add to Zero Extended XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi[j]</td>
<td>P1</td>
<td>SR</td>
<td>79</td>
<td></td>
<td>Add to Zero Extended &amp; record OV XO-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi[j]</td>
<td>P1</td>
<td>100</td>
<td>AND X-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi[j]</td>
<td>P1</td>
<td>SR</td>
<td>101</td>
<td></td>
<td>AND with Complement X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi0[j]</td>
<td>P1</td>
<td>SR</td>
<td>99</td>
<td></td>
<td>AND Immediate D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi[j]</td>
<td>P1</td>
<td>SR</td>
<td>99</td>
<td></td>
<td>AND Immediate Shifted D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi[j]</td>
<td>P1</td>
<td></td>
<td>41</td>
<td>Branch I-form</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdir[a]</td>
<td>P1</td>
<td>CT</td>
<td>41</td>
<td></td>
<td>Branch Conditional B-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdir[a]</td>
<td>P1</td>
<td>CT</td>
<td>42</td>
<td></td>
<td>Branch Conditional to Count Register XL-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v2.07</td>
<td>478</td>
<td>Decimal Add Modulo VX-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v3.0</td>
<td>480</td>
<td>Decimal Convert From National VX-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v3.0</td>
<td>485</td>
<td>Decimal Convert From Signed Quadword VX-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v3.0</td>
<td>481</td>
<td>Decimal Convert From Zoned VX-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v3.0</td>
<td>489</td>
<td>Decimal Copy Sign VX-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v3.0</td>
<td>483</td>
<td>Decimal Convert To National VX-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v3.0</td>
<td>486</td>
<td>Decimal Convert To Signed Quadword VX-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v3.0</td>
<td>484</td>
<td>Decimal Convert To Zoned VX-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v3.0</td>
<td>481</td>
<td>Decimal Shift VX-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v3.0</td>
<td>490</td>
<td>Decimal Set Sign VX-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v3.0</td>
<td>493</td>
<td>Decimal Shift and Round VX-form</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdirad</td>
<td>v2.07</td>
<td>478</td>
<td>Decimal Subtract Modulo VX-form</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 1 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compilatory Subsets(^2)</th>
<th>Linux Optional Category(^3)</th>
<th>Always Optional Category(^4)</th>
<th>Mnemonic</th>
<th>Version(^5)</th>
<th>Privilege(^6)</th>
<th>Mode Dep(^7)</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>10011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdtrunc.</td>
<td>v3.0</td>
<td>CT</td>
<td></td>
<td>484</td>
<td>Decimal Truncate VX-form</td>
</tr>
<tr>
<td>10011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdus.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>492</td>
<td>Decimal Unsigned Shift VX-form</td>
</tr>
<tr>
<td>10011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcdtrunc.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>495</td>
<td>Decimal Unsigned Truncate VX-form</td>
</tr>
<tr>
<td>10001010111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcd1r[]</td>
<td>P1</td>
<td>CT</td>
<td></td>
<td>42</td>
<td>Branch Conditional to Link Register X-form</td>
</tr>
<tr>
<td>00011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcmpd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>105</td>
<td>Bit Permuted Doubleword X-form</td>
</tr>
<tr>
<td>10011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcmp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>119</td>
<td>Byte-Reverse Doubleword X-form</td>
</tr>
<tr>
<td>10011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcmpb</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>119</td>
<td>Byte-Reverse Halfword X-form</td>
</tr>
<tr>
<td>10011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>brw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>119</td>
<td>Byte-Reverse Word X-form</td>
</tr>
<tr>
<td>10011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cbcdld</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>117</td>
<td>Convert Binary Coded Decimal To Declets X-form</td>
</tr>
<tr>
<td>10011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cbdbcd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>117</td>
<td>Convert Declets To Binary Coded Decimal X-form</td>
</tr>
<tr>
<td>10011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>clfreep</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>106</td>
<td>Centrifuge Doubleword X-form</td>
</tr>
<tr>
<td>10011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmph</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Clear BRHBR Doubleword X-form</td>
</tr>
<tr>
<td>00011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmp</td>
<td>P1</td>
<td>93</td>
<td></td>
<td></td>
<td>Compare X-form</td>
</tr>
<tr>
<td>00011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmpeqb</td>
<td>v3.0</td>
<td>95</td>
<td></td>
<td></td>
<td>Compare Equal Byte X-form</td>
</tr>
<tr>
<td>00011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmpe</td>
<td>P1</td>
<td>93</td>
<td></td>
<td></td>
<td>Compare Immediate D-form</td>
</tr>
<tr>
<td>00011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmpl</td>
<td>P1</td>
<td>93</td>
<td></td>
<td></td>
<td>Compare Logical X-form</td>
</tr>
<tr>
<td>00011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmpli</td>
<td>P1</td>
<td>93</td>
<td></td>
<td></td>
<td>Compare Logical Immediate D-form</td>
</tr>
<tr>
<td>00011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmrpb</td>
<td>v3.0</td>
<td>94</td>
<td></td>
<td></td>
<td>Compare Ranged Byte X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cntldz</td>
<td>PPC</td>
<td>SR</td>
<td>104</td>
<td></td>
<td>Count Leading Zeros Doubleword X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cntlzdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>105</td>
<td>Count Leading Zeros Doubleword under bit Mask X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cntlzw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>102</td>
<td>Count Leading Zeros Word X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cntlzdm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>102</td>
<td>Count Trailing Zeros Doubleword X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cntlz</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>105</td>
<td>Count Trailing Zeros Word under bit Mask X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cntlzw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>102</td>
<td>Count Trailing Zora Word X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>copy</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>1068</td>
<td>Copy X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cpab3ort</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>1069</td>
<td>Copy-Paste Abort X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>crand</td>
<td>P1</td>
<td>44</td>
<td></td>
<td></td>
<td>Condition Register AND X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>crandc</td>
<td>P1</td>
<td>45</td>
<td></td>
<td></td>
<td>Condition Register AND with Complement X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cneqv</td>
<td>P1</td>
<td>45</td>
<td></td>
<td></td>
<td>Condition Register Equivalent X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmart</td>
<td>P1</td>
<td>44</td>
<td></td>
<td></td>
<td>Condition Register NAND X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cmor</td>
<td>P1</td>
<td>45</td>
<td></td>
<td></td>
<td>Condition Register NOR X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cnor</td>
<td>P1</td>
<td>44</td>
<td></td>
<td></td>
<td>Condition Register OR X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cnorc</td>
<td>P1</td>
<td>45</td>
<td></td>
<td></td>
<td>Condition Register OR with Complement X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cncx</td>
<td>P1</td>
<td>44</td>
<td></td>
<td></td>
<td>Condition Register XOR X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>daddl</td>
<td>v2.05</td>
<td>206</td>
<td></td>
<td></td>
<td>DPF Add X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>daddq</td>
<td>v2.05</td>
<td>206</td>
<td></td>
<td></td>
<td>DPF Add Quad X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>darr</td>
<td>v3.0</td>
<td>86</td>
<td></td>
<td></td>
<td>Deliver A Random Number X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcdfb</td>
<td>PPC</td>
<td>1064</td>
<td></td>
<td></td>
<td>Data Cache Block Flush X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcdbst</td>
<td>PPC</td>
<td>1063</td>
<td></td>
<td></td>
<td>Data Cache Block Store X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcdbt</td>
<td>PPC</td>
<td>1061</td>
<td></td>
<td></td>
<td>Data Cache Block Touch X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcdbstb</td>
<td>PPC</td>
<td>1062</td>
<td></td>
<td></td>
<td>Data Cache Block Touch for Store X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcdbz</td>
<td>P1</td>
<td>1063</td>
<td></td>
<td></td>
<td>Data Cache Block set to Zero X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcflix</td>
<td>v2.06</td>
<td>231</td>
<td></td>
<td></td>
<td>DPF Convert From Fixed X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dcflixq</td>
<td>v2.05</td>
<td>231</td>
<td></td>
<td></td>
<td>DPF Convert From Fixed Quad X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dfclxq</td>
<td>v3.1</td>
<td>232</td>
<td></td>
<td></td>
<td>DPF Convert From Fixed Quadword X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dfclpaq</td>
<td>v2.05</td>
<td>272</td>
<td></td>
<td></td>
<td>DPF Compare Ordered X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dfclpa</td>
<td>v2.05</td>
<td>272</td>
<td></td>
<td></td>
<td>DPF Compare Ordered Quad X-form</td>
</tr>
<tr>
<td>01011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dfclpu</td>
<td>v2.05</td>
<td>271</td>
<td></td>
<td></td>
<td>DPF Compare Uplimited X-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 2 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>211</td>
<td>DFP Compare Unordered Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>229</td>
<td>DFP Convert To DFP Long X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dfcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dfcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
<tr>
<td>dfp</td>
<td></td>
<td>dcmpeqq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>233</td>
<td>DFP Convert To Fixed Quad X-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 3 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subset</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstdc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Data Class Z22-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstdcq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Data Class Quad Z22-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstdg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Data Group Z22-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstdgq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Data Group Quad Z22-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstex</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Exponent X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstexq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Exponent Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Immediate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstdx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Immediate Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Immediate Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstex</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Exponent X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstexq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Exponent Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Immediate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Immediate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dtstsfq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFP Test Significance Quad X-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 4 of 30)
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Mode Dep2</th>
<th>Privilege6</th>
<th>Version5</th>
<th>Mnemonic</th>
<th>Linux Optional Category3</th>
<th>Always Optional Category4</th>
<th>Compliancy Subsets2</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmsrgow</td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>162</td>
<td>Floating Merge Odd Word X-form</td>
</tr>
<tr>
<td>fmsubs[.]</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>168</td>
<td>Floating Multiply-Subtract A-form</td>
</tr>
<tr>
<td>fmsub[.]</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>168</td>
<td>Floating Multiply-Subtract Single A-form</td>
</tr>
<tr>
<td>fnmul[.]</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>164</td>
<td>Floating Multiply A-form</td>
</tr>
<tr>
<td>fnms[.]</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>164</td>
<td>Floating Multiply Single A-form</td>
</tr>
<tr>
<td>fnabs[.]</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>161</td>
<td>Floating Negative Absolute Value X-form</td>
</tr>
<tr>
<td>fneg[.]</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>161</td>
<td>Floating Negate X-form</td>
</tr>
<tr>
<td>fnmadd[.]</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Add A-form</td>
</tr>
<tr>
<td>fnmadd[.]</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Add Single A-form</td>
</tr>
<tr>
<td>fnmsub[.]</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Subtract A-form</td>
</tr>
<tr>
<td>fnmsub[.]</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>169</td>
<td>Floating Negative Multiply-Subtract Single A-form</td>
</tr>
<tr>
<td>fdiv[.]</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>166</td>
<td>Floating Reciprocal Test A-form</td>
</tr>
<tr>
<td>fnsqrt[.]</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>176</td>
<td>Floating Round to Integer Minus X-form</td>
</tr>
<tr>
<td>fnsqrt[.]</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>176</td>
<td>Floating Round to Integer Nearest X-form</td>
</tr>
<tr>
<td>fnsqrt[.]</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>178</td>
<td>Floating Round to Integer Plus X-form</td>
</tr>
<tr>
<td>fnsqrt[.]</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>178</td>
<td>Floating Round to Integer Toward Zero X-form</td>
</tr>
<tr>
<td>fn sqrt[.]</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>170</td>
<td>Floating Round to Single-Precision X-form</td>
</tr>
<tr>
<td>fsqrt[.]</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>166</td>
<td>Floating Reciprocal Square Root Estimate A-form</td>
</tr>
<tr>
<td>fsqrt[.]</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>166</td>
<td>Floating Reciprocal Square Root Estimate Single A-form</td>
</tr>
<tr>
<td>fselect[.]</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>180</td>
<td>Floating Select A-form</td>
</tr>
<tr>
<td>fselect[.]</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>165</td>
<td>Floating Square Root A-form</td>
</tr>
<tr>
<td>fselect[.]</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>165</td>
<td>Floating Square Root Single A-form</td>
</tr>
<tr>
<td>fselect[.]</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>163</td>
<td>Floating Subtract A-form</td>
</tr>
<tr>
<td>fselect[.]</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>163</td>
<td>Floating Subtract Single A-form</td>
</tr>
<tr>
<td>fldiv[.]</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>166</td>
<td>Floating Test for software Divide X-form</td>
</tr>
<tr>
<td>fmsqrt[.]</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>167</td>
<td>Floating Test for software Square Root X-form</td>
</tr>
<tr>
<td>fhrd</td>
<td></td>
<td>v2.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1152</td>
<td>Return From Interrupt Doubleword Hypervisor XL-form</td>
</tr>
<tr>
<td>fcbi</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1052</td>
<td>Instruction Cache Block Invalidate X-form</td>
</tr>
<tr>
<td>fcbi</td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1052</td>
<td>Instruction Cache Block Touch X-form</td>
</tr>
<tr>
<td>isel</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>98</td>
<td>Integer Select A-form</td>
</tr>
<tr>
<td>isync</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1076</td>
<td>Instruction Synchronize XL-form</td>
</tr>
<tr>
<td>ifbax</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1077</td>
<td>Load Byte And Reserve Indexed X-form</td>
</tr>
<tr>
<td>ifbax</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>52</td>
<td>Load Byte and Zero D-form</td>
</tr>
<tr>
<td>ifz</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>52</td>
<td>Load Byte and Zero D-form</td>
</tr>
<tr>
<td>ifz</td>
<td></td>
<td>v2.05</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1164</td>
<td>Load Byte &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td>ifzbz</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>52</td>
<td>Load Byte and Zero with Update D-form</td>
</tr>
<tr>
<td>ifzbz</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>52</td>
<td>Load Byte and Zero with Update Indexed X-form</td>
</tr>
<tr>
<td>ifbxz</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>52</td>
<td>Load Byte and Zero Indexed X-form</td>
</tr>
<tr>
<td>ifbxz</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>57</td>
<td>Load Doubleword DS-form</td>
</tr>
<tr>
<td>id</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1062</td>
<td>Load Doubleword And Reserve Indexed X-form</td>
</tr>
<tr>
<td>idarx</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1073</td>
<td>Load Doubleword Atomic X-form</td>
</tr>
<tr>
<td>idarx</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69</td>
<td>Load Doubleword Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td>idbx</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1164</td>
<td>Load Doubleword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td>idbx</td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>57</td>
<td>Load Doubleword with Update DS-form</td>
</tr>
<tr>
<td>idbx</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>57</td>
<td>Load Doubleword with Update Indexed X-form</td>
</tr>
<tr>
<td>idbx</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>57</td>
<td>Load Doubleword Indexed X-form</td>
</tr>
<tr>
<td>idbx</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>152</td>
<td>Load Floating-Point Double D-form</td>
</tr>
<tr>
<td>idbx</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>152</td>
<td>Load Floating-Point Double Indexed X-form</td>
</tr>
<tr>
<td>idbx</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>152</td>
<td>Load Floating-Point Double with Update D-form</td>
</tr>
<tr>
<td>idbx</td>
<td></td>
<td>PPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>152</td>
<td>Load Floating-Point Double with Update Indexed X-form</td>
</tr>
<tr>
<td>idbx</td>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>152</td>
<td>Load Floating-Point Double Indexed X-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 5 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep.</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ifiawx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>153</td>
<td>Load Floating-Point as Integer Word &amp; Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ifiawx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>153</td>
<td>Load Floating-Point as Integer Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ffs</td>
<td>P1</td>
<td></td>
<td></td>
<td>150</td>
<td>Load Floating-Point Single D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ffsu</td>
<td>P1</td>
<td></td>
<td></td>
<td>150</td>
<td>Load Floating-Point Single with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fflux</td>
<td>P1</td>
<td></td>
<td></td>
<td>151</td>
<td>Load Floating-Point Single with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fflx</td>
<td>P1</td>
<td></td>
<td></td>
<td>150</td>
<td>Load Floating-Point Single Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhax</td>
<td>P1</td>
<td></td>
<td></td>
<td>54</td>
<td>Load Halfword D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhau</td>
<td>P1</td>
<td></td>
<td></td>
<td>1078</td>
<td>Load Halfword And Reserve Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhaux</td>
<td>P1</td>
<td></td>
<td></td>
<td>54</td>
<td>Load Halfword Algebraic with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhax</td>
<td>P1</td>
<td></td>
<td></td>
<td>54</td>
<td>Load Halfword Algebraic Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhbx</td>
<td>P1</td>
<td></td>
<td></td>
<td>67</td>
<td>Load Halfword Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lh</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td>Load Halfword and Zero D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhczx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>1164</td>
<td>Load Halfword &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhcz</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td>Load Halfword and Zero with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhczux</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td>Load Halfword and Zero with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhcz</td>
<td>P1</td>
<td></td>
<td></td>
<td>53</td>
<td>Load Halfword and Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lhcz</td>
<td>P1</td>
<td></td>
<td></td>
<td>70</td>
<td>Load Multiple Word D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lq</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>65</td>
<td>Load Quadword D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lqarx</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Load Quadword And Reserve Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lqm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>Load Quadword Metadata</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lswi</td>
<td>P1</td>
<td></td>
<td></td>
<td>72</td>
<td>Load String Word Immediate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lswx</td>
<td>P1</td>
<td></td>
<td></td>
<td>72</td>
<td>Load String Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvbx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>268</td>
<td>Load Vector Element Byte Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvbx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>269</td>
<td>Load Vector Element Halfword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvbx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>270</td>
<td>Load Vector Element Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lv</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>277</td>
<td>Load Vector for Shift Left Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lv</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>277</td>
<td>Load Vector for Shift Right Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lv</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>271</td>
<td>Load Vector Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lv</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>271</td>
<td>Load Vector Indexed Last X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwa</td>
<td>PPC</td>
<td></td>
<td></td>
<td>56</td>
<td>Load Word Arithmetic DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwarx</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1078</td>
<td>Load Word &amp; Reserve Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwar</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1078</td>
<td>Load Word Arithmetic DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwar</td>
<td>PPC</td>
<td></td>
<td></td>
<td>56</td>
<td>Load Word Arithmetic Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwar</td>
<td>PPC</td>
<td></td>
<td></td>
<td>56</td>
<td>Load Word Arithmetic Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwar</td>
<td>PPC</td>
<td></td>
<td></td>
<td>56</td>
<td>Load Word Arithmetic Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwr</td>
<td>PPC</td>
<td></td>
<td></td>
<td>68</td>
<td>Load Word Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwr</td>
<td>P1</td>
<td></td>
<td></td>
<td>55</td>
<td>Load Word and Zero D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwoczx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>1164</td>
<td>Load Word &amp; Zero Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwocz</td>
<td>P1</td>
<td></td>
<td></td>
<td>55</td>
<td>Load Word and Zero with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwocz</td>
<td>P1</td>
<td></td>
<td></td>
<td>55</td>
<td>Load Word and Zero with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwocz</td>
<td>P1</td>
<td></td>
<td></td>
<td>55</td>
<td>Load Word and Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwsd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>610</td>
<td>Load V8X Scalar Doubleword DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwsd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>611</td>
<td>Load V8X Scalar Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwsibz</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>612</td>
<td>Load V8X Scalar as Integer Byte &amp; Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwsibz</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>612</td>
<td>Load V8X Scalar as Integer Halfword &amp; Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwsibz</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>613</td>
<td>Load V8X Scalar as Integer Word Algebric Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwsibzw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>614</td>
<td>Load V8X Scalar as Integer Word &amp; Zero Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxsisp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>615</td>
<td>Load V8X Scalar Single-Precision DS-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 6 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Compliancy Subsets2</th>
<th>Linux Optional Category3</th>
<th>Always Optional Category4</th>
<th>Mnemonic</th>
<th>Version5</th>
<th>Privilege6</th>
<th>Mode Dep7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lsspdx</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>616</td>
<td>Load VSX Scalar Single-Precision Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>617</td>
<td>Load VSX Vector DQ-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwvb16x</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>618</td>
<td>Load VSX Vector Byte/16 Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwvd2x</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>619</td>
<td>Load VSX Vector Doubleword/2 Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxvdhx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>633</td>
<td>Load VSX Vector Doubleword &amp; Splat Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwv8x</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>634</td>
<td>Load VSX Vector Halfword*8 Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvkq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>620</td>
<td>Load VSX Vector Special Value Quadword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvvp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>621</td>
<td>Load VSX Vector with Length X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>623</td>
<td>Load VSX Vector with Length Left-justified X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvvpx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>625</td>
<td>Load VSX Vector Paired DQ-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwrvbxx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>627</td>
<td>Load VSX Vector Rightmost Byte Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwrvdx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>628</td>
<td>Load VSX Vector Rightmost Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lvrvx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>630</td>
<td>Load VSX Vector Rightmost Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwrv4x</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>635</td>
<td>Load VSX Vector Word*4 Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lwvwx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>636</td>
<td>Load VSX Vector Word &amp; Splat Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lxv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>631</td>
<td>Load VSX Vector Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>maddhdh</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>88</td>
<td>Multiply-Add High Doubleword VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>maddhdhu</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>88</td>
<td>Multiply-Add High Doubleword Unsigned VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>maddld</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>88</td>
<td>Multiply-Add Low Doubleword VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mcrf</td>
<td>P1</td>
<td></td>
<td></td>
<td>46</td>
<td>Move Condition Register Field X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mcrf[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>182</td>
<td>Move From Condition Register from FPSCR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfcrf</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>128</td>
<td>Move From CR from XER Extended X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfcr</td>
<td>P1</td>
<td></td>
<td></td>
<td>128</td>
<td>Move From Condition Register XF-X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfcrf[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>182</td>
<td>Move From FPSCR X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfssdri</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>183</td>
<td>Move From FPSCR Control &amp; Set DRN Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfssdri</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>183</td>
<td>Move From FPSCR Control &amp; Set DRN X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfsscr</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>183</td>
<td>Move From FPSCR Control &amp; Set RN X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfsscr</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>183</td>
<td>Move From FPSCR Control &amp; Set RN Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfss</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>183</td>
<td>Move From FPSCR Lightweight X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfssr</td>
<td>P1</td>
<td></td>
<td></td>
<td>1176</td>
<td>Move From MSR X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfocrf</td>
<td>v2.01</td>
<td></td>
<td></td>
<td>128</td>
<td>Move From One Condition Register Field XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfspcr</td>
<td>P1</td>
<td>O</td>
<td></td>
<td>126</td>
<td>Move From Special Purpose Register XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfspcr</td>
<td>P1</td>
<td></td>
<td></td>
<td>1173</td>
<td>Move From Special Purpose Register XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfsb</td>
<td>PPC</td>
<td></td>
<td></td>
<td>1094</td>
<td>Move From Time Base XFX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfsvcr</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>496</td>
<td>Move From Vector Status and Control Register VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfsvrd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>120</td>
<td>Move From VSR Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfsr</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>120</td>
<td>Move From VSR Lower Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mfiswz</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>121</td>
<td>Move From VSR Word and Zero X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>modsd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>91</td>
<td>Modulo Signed Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>modsdw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>85</td>
<td>Modulo Signed Word X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>modud</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>91</td>
<td>Modulo Unsigned Doubleword X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>modudw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>85</td>
<td>Modulo Unsigned Word X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msgdr</td>
<td>v2.07</td>
<td>HV</td>
<td></td>
<td>1329</td>
<td>Message Clear X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msgdrp</td>
<td>v2.07</td>
<td>P</td>
<td></td>
<td>1331</td>
<td>Message Clear Privileged X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msgdru</td>
<td>v3.0C</td>
<td>UV</td>
<td></td>
<td>1328</td>
<td>Ultravisor Message Clear X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msgdand</td>
<td>v2.07</td>
<td>HV</td>
<td></td>
<td>1328</td>
<td>Message Send X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msgdandp</td>
<td>v2.07</td>
<td>P</td>
<td></td>
<td>1330</td>
<td>Message Send Privileged X-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 7 of 30)

Appendix H. Power ISA Instruction Set Sorted by Mnemonic
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Book</th>
<th>Compliancy Subsets2</th>
<th>Linux Optional Category3</th>
<th>Always Optional Category4</th>
<th>Mnemonic</th>
<th>Version5</th>
<th>Privilege6</th>
<th>Mode Dep7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>magndu</td>
<td>v3.0C</td>
<td>UV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>micrsync</td>
<td>P1</td>
<td>127</td>
<td>Move To Condition Register Fields XFX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mfsb[0()]</td>
<td>P1</td>
<td>185</td>
<td>Move To FPSCR Bit 0 X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mfsb[1()]</td>
<td>P1</td>
<td>184</td>
<td>Move To FPSCR Fields XFL-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtcrf</td>
<td>PPC</td>
<td>1174</td>
<td>Move To MSR X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtfsb0[.]</td>
<td>P1</td>
<td>184</td>
<td>Move To FPSCR Bit 0 X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtfsb1[.]</td>
<td>P1</td>
<td>184</td>
<td>Move To FPSCR Fields XFL-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtfsf[.]</td>
<td>P1</td>
<td>184</td>
<td>Move To FPSCR Field Immediate X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtmsr</td>
<td>PPC</td>
<td>1175</td>
<td>Move To MSR Doubleword X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtmsrd</td>
<td>PPC</td>
<td>1175</td>
<td>Move To MSR Doubleword X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvscr</td>
<td>v2.01</td>
<td>127</td>
<td>Move To One Condition Register Field XFX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvscrm</td>
<td>v3.1</td>
<td>483</td>
<td>Move To VSR Byte Mask VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvscrmn</td>
<td>v3.1</td>
<td>483</td>
<td>Move To VSR Byte Mask Immediate DX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvscrd</td>
<td>v3.0</td>
<td>123</td>
<td>Move To VSR Doubleword X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvscrdm</td>
<td>v3.1</td>
<td>464</td>
<td>Move To VSR Doubleword Mask VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvsvms</td>
<td>v3.1</td>
<td>483</td>
<td>Move To VSR Halfword Mask VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvsvmsm</td>
<td>v3.1</td>
<td>483</td>
<td>Move To VSR Halfword Mask VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvsvsm</td>
<td>v3.1</td>
<td>483</td>
<td>Move To VSR Quadword Mask VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvswwa</td>
<td>v2.07</td>
<td>122</td>
<td>Move To VSR Word Algebraic X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvswwm</td>
<td>v3.1</td>
<td>464</td>
<td>Move To VSR Word Mask VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvswwm</td>
<td>v3.1</td>
<td>464</td>
<td>Move To VSR Word Mask VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtvswwm</td>
<td>v3.1</td>
<td>464</td>
<td>Move To VSR Word Mask VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multd[.]</td>
<td>PPC</td>
<td>87</td>
<td>Multiply High Doubleword XO-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multdu[.]</td>
<td>PPC</td>
<td>87</td>
<td>Multiply High Doubleword Unsigned XO-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multhd[.]</td>
<td>PPC</td>
<td>81</td>
<td>Multiply High Word XO-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multlw[.]</td>
<td>PPC</td>
<td>81</td>
<td>Multiply High Word Unsigned XO-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multld[.]</td>
<td>PPC</td>
<td>87</td>
<td>Multiply Low Doubleword XO-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multl[.]</td>
<td>PPC</td>
<td>87</td>
<td>Multiply Low Doubleword &amp; record OV XO-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul[.]</td>
<td>P1</td>
<td>81</td>
<td>Multiply Low Immediate D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul[.]</td>
<td>P1</td>
<td>81</td>
<td>Multiply Low Word XO-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul[.]</td>
<td>P1</td>
<td>81</td>
<td>Multiply Low Word &amp; record OV XO-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nand[.]</td>
<td>P1</td>
<td>100</td>
<td>NAND X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>neg[.]</td>
<td>P1</td>
<td>80</td>
<td>Negate XO-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nego[.]</td>
<td>P1</td>
<td>80</td>
<td>Negate &amp; record OV XO-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nior[.]</td>
<td>P1</td>
<td>101</td>
<td>NOR X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or[.]</td>
<td>P1</td>
<td>101</td>
<td>OR X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>orc[.]</td>
<td>P1</td>
<td>101</td>
<td>OR with Complement X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or[.]</td>
<td>P1</td>
<td>99</td>
<td>OR Immediate D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or[.]</td>
<td>P1</td>
<td>100</td>
<td>OR Immediate Shifted D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>paddi</td>
<td>v3.1</td>
<td>76</td>
<td>Prefixed Add Immediate MLS:D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>past[.]</td>
<td>v3.0</td>
<td>1068</td>
<td>Paste X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pdepd</td>
<td>v3.1</td>
<td>106</td>
<td>Parallel Bits Deposit Doubleword X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pextd</td>
<td>v3.1</td>
<td>106</td>
<td>Parallel Bits Extract Doubleword X-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>plbd</td>
<td>v3.1</td>
<td>52</td>
<td>Prefixed Load Byte and Zero MLS:D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>plbd</td>
<td>v3.1</td>
<td>57</td>
<td>Prefixed Load Doubleword MLS:D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>plbd</td>
<td>v3.1</td>
<td>152</td>
<td>Prefixed Load Floating-Point Double MLS:D-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 8 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pfls</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>150</td>
<td>Prefixed Load Floating-Point Single MLS.D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plha</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>54</td>
<td>Prefixed Load Halfword Algebraic MLS.D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plhz</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>53</td>
<td>Prefixed Load Halfword and Zero MLS.D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>65</td>
<td>Prefixed Load Quadword MLS.D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plwa</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>56</td>
<td>Prefixed Load Word Algebraic MLS.D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plwz</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>55</td>
<td>Prefixed Load Word and Zero MLS.D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>plxsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>610</td>
<td>Prefixed Load VSX Doubleword MLS.D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvb16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvb16ger2mm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvb16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvb16ger2pm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvb16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>827</td>
<td>Prefixed Masked VSX Vector bfloat16 GER (Rank-2 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvb16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvb16ger2n</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvb16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvb16ger2pm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvf32ger</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvf32germ</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvf32gemp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMA</td>
<td>MMA</td>
<td></td>
<td>pmxvf32gerpn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 9 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Complimentary Subsets²</th>
<th>Linux Optional Category³</th>
<th>Always Optional Category⁴</th>
<th>Mnemonic</th>
<th>Version⁵</th>
<th>Privilege⁶</th>
<th>Mode Dep⁷</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvf32gerpp</td>
<td>v3.1</td>
<td>875</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvf64ger</td>
<td>v3.1</td>
<td>879</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvf64germm</td>
<td>v3.1</td>
<td>879</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvf64germp</td>
<td>v3.1</td>
<td>879</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvi16ger</td>
<td>v3.1</td>
<td>891</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvi16ger2</td>
<td>v3.1</td>
<td>891</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvi16ger2pp</td>
<td>v3.1</td>
<td>893</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvi16ger2s</td>
<td>v3.1</td>
<td>893</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvi16ger2spp</td>
<td>v3.1</td>
<td>893</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvi4ger8</td>
<td>v3.1</td>
<td>883</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvi4ger8pp</td>
<td>v3.1</td>
<td>883</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvi8ger4</td>
<td>v3.1</td>
<td>886</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvi8ger4pp</td>
<td>v3.1</td>
<td>886</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pmxvi8ger4spp</td>
<td>v3.1</td>
<td>889</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pnp</td>
<td>v3.1</td>
<td>130</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>popcntb</td>
<td>v2.02</td>
<td>103</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>popcntd</td>
<td>v2.06</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>popcntwb</td>
<td>v2.06</td>
<td>103</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>prlyd</td>
<td>v2.05</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>prlyw</td>
<td>v2.05</td>
<td>103</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>pstd</td>
<td>v3.1</td>
<td>62</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>psfld</td>
<td>v3.1</td>
<td>157</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>psftd</td>
<td>v3.1</td>
<td>155</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101 11100</td>
<td>1</td>
<td>MAA</td>
<td>psft</td>
<td>v3.1</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 10 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliance Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>psiq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>66</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>psiw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>61</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>psixsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>638</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>psixssp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>642</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>psixv</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>644</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>psiv</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>654</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rlebb</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>1152</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rfid</td>
<td>PPC P</td>
<td></td>
<td></td>
<td>1151</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rsfcv</td>
<td>PPC P</td>
<td></td>
<td></td>
<td>111</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ridc[]</td>
<td>PPC SR</td>
<td>111</td>
<td>111</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ridir[]</td>
<td>PPC SR</td>
<td>112</td>
<td>112</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ridic[]</td>
<td>PPC SR</td>
<td>111</td>
<td>111</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ridic[]</td>
<td>PPC SR</td>
<td>110</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ridic[]</td>
<td>PPC SR</td>
<td>112</td>
<td>112</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nrmw[]</td>
<td>P1 SR</td>
<td>108</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nrmw[]</td>
<td>P1 SR</td>
<td>107</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nrmw[]</td>
<td>P1 SR</td>
<td>106</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sc</td>
<td>PPC</td>
<td>47</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>scv</td>
<td>v3.0</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setb</td>
<td>v3.0</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbc</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setbcr</td>
<td>v3.1</td>
<td></td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC SR</td>
<td>115</td>
<td>115</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>slf[]</td>
<td>PPC</td>
<td>1229</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 11 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbcx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Byte Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbcx</td>
<td>v2.06</td>
<td>P1</td>
<td></td>
<td>1079</td>
<td>Store Byte Conditional Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbu</td>
<td>P1</td>
<td></td>
<td></td>
<td>59</td>
<td>Store Byte with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbux</td>
<td>P1</td>
<td></td>
<td></td>
<td>59</td>
<td>Store Byte with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbx</td>
<td>P1</td>
<td></td>
<td></td>
<td>59</td>
<td>Store Byte Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>std</td>
<td>PPC</td>
<td>62</td>
<td></td>
<td></td>
<td>Store Doubleword DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdat</td>
<td>v3.0</td>
<td>1075</td>
<td></td>
<td></td>
<td>Store Doubleword Atomic X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdbrx</td>
<td>v2.06</td>
<td>69</td>
<td></td>
<td></td>
<td>Store Doubleword Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdctx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Doubleword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdcx</td>
<td>PPC</td>
<td>1082</td>
<td></td>
<td></td>
<td>Store Doubleword Conditional Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdu</td>
<td>PPC</td>
<td>62</td>
<td></td>
<td></td>
<td>Store Doubleword with Update DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdux</td>
<td>PPC</td>
<td>63</td>
<td></td>
<td></td>
<td>Store Doubleword with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdx</td>
<td>PPC</td>
<td>62</td>
<td></td>
<td></td>
<td>Store Doubleword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdf</td>
<td>P1</td>
<td>157</td>
<td></td>
<td></td>
<td>Store Floating-Point Double D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdfp</td>
<td>v2.05</td>
<td>160</td>
<td></td>
<td></td>
<td>Store Floating-Point Double Pair DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdfpdx</td>
<td>v2.05</td>
<td>160</td>
<td></td>
<td></td>
<td>Store Floating-Point Double Pair Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdfu</td>
<td>P1</td>
<td>157</td>
<td></td>
<td></td>
<td>Store Floating-Point Double with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdlux</td>
<td>P1</td>
<td>158</td>
<td></td>
<td></td>
<td>Store Floating-Point Double Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stdfux</td>
<td>PPC</td>
<td>158</td>
<td></td>
<td></td>
<td>Store Floating-Point as Integer Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sfs</td>
<td>P1</td>
<td>155</td>
<td></td>
<td></td>
<td>Store Floating-Point Single D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sfsu</td>
<td>P1</td>
<td>155</td>
<td></td>
<td></td>
<td>Store Floating-Point Single with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sfsux</td>
<td>P1</td>
<td>156</td>
<td></td>
<td></td>
<td>Store Floating-Point Single with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sfx</td>
<td>P1</td>
<td>155</td>
<td></td>
<td></td>
<td>Store Floating-Point Single Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sfx</td>
<td>P1</td>
<td>60</td>
<td></td>
<td></td>
<td>Store Halfword D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Halfword Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbxxx</td>
<td>v2.06</td>
<td>1080</td>
<td></td>
<td></td>
<td>Store Halfword Conditional Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbu</td>
<td>P1</td>
<td>60</td>
<td></td>
<td></td>
<td>Store Halfword with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbux</td>
<td>P1</td>
<td>60</td>
<td></td>
<td></td>
<td>Store Halfword with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sbx</td>
<td>P1</td>
<td>60</td>
<td></td>
<td></td>
<td>Store Halfword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stmw</td>
<td>P1</td>
<td>70</td>
<td></td>
<td></td>
<td>Store Multiple Word D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stop</td>
<td>v3.0</td>
<td>P</td>
<td>1155</td>
<td></td>
<td>Stop XL-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stq</td>
<td>v2.03</td>
<td>66</td>
<td></td>
<td></td>
<td>Store Quadword DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stqcx</td>
<td>v2.07</td>
<td>73</td>
<td></td>
<td></td>
<td>Store Quadword Conditional Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>staw</td>
<td>P1</td>
<td>73</td>
<td></td>
<td></td>
<td>Store String Word Immediate X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stawx</td>
<td>P1</td>
<td>73</td>
<td></td>
<td></td>
<td>Store String Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvbx</td>
<td>v2.03</td>
<td>272</td>
<td></td>
<td></td>
<td>Store Vector Element Byte Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvctx</td>
<td>v2.03</td>
<td>273</td>
<td></td>
<td></td>
<td>Store Vector Element Halfword Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvwx</td>
<td>v2.03</td>
<td>274</td>
<td></td>
<td></td>
<td>Store Vector Element Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stv</td>
<td>v2.03</td>
<td>275</td>
<td></td>
<td></td>
<td>Store Vector Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stvnl</td>
<td>v2.03</td>
<td>275</td>
<td></td>
<td></td>
<td>Store Vector Indexed Last X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stw</td>
<td>P1</td>
<td>61</td>
<td></td>
<td></td>
<td>Store Word D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stwatt</td>
<td>v3.0</td>
<td>1075</td>
<td></td>
<td></td>
<td>Store Word Atomic X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stwbx</td>
<td>P1</td>
<td>68</td>
<td></td>
<td></td>
<td>Store Word Byte-Reverse Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stwctx</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>1165</td>
<td>Store Word Caching Inhibited Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stwcx</td>
<td>PPC</td>
<td>1081</td>
<td></td>
<td></td>
<td>Store Word Conditional Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stwu</td>
<td>P1</td>
<td>61</td>
<td></td>
<td></td>
<td>Store Word with Update D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stwx</td>
<td>P1</td>
<td>61</td>
<td></td>
<td></td>
<td>Store Word with Update Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stwx</td>
<td>P1</td>
<td>61</td>
<td></td>
<td></td>
<td>Store Word Indexed X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stx</td>
<td>v3.0</td>
<td>638</td>
<td></td>
<td></td>
<td>Store VSX Scalar Doubleword DS-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stxsdx</td>
<td>v2.06</td>
<td>639</td>
<td></td>
<td></td>
<td>Store VSX Scalar Doubleword Indexed X-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 12 of 30)
## Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 13 of 30)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vaddsbs v2.03 323</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Signed Byte Saturate VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddfp v2.03 422</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Floating-Point VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddeuqm v2.07 329</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended Unsigned Quadword Modulo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddcuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add &amp; write Carry Unsigned Quadword VX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vabsduw v3.0 379</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Absolute Difference Unsigned Word VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vabsduh v3.0 378</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Absolute Difference Unsigned Halfword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vabsdub v3.0 378</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Absolute Difference Unsigned Byte VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>urtid v3.0C UV 1153</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ultravisor Return From Interrupt Doubleword XL-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vabsdf v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Absolute Difference Floating-Point VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddecuq v2.07 330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword VX-form</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Instruction</td>
<td>Compliancy Subsets</td>
<td>Linux Optional Category</td>
<td>Always Optional Category</td>
<td>Mnemonic</td>
<td>Version</td>
<td>Privilege</td>
<td>Mode Dep</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>---------------------</td>
<td>------------------------</td>
<td>-------------------------</td>
<td>----------</td>
<td>---------</td>
<td>-----------</td>
<td>----------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaddsws</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>324</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaddubm</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>325</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaddubs</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>327</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaddudm</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>326</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vadduh</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>325</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vadduhs</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>327</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaddugm</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>329</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaddumx</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>326</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vadduws</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>328</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vand</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>402</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vandc</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>402</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaavgb</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>375</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaavgh</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>376</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaavgw</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>377</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaavgb</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>375</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaavgh</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>376</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vaavgw</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>377</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vbpermmd</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>461</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vbpermq</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>462</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcfx</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>425</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcfuged</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>456</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcfux</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>425</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcipher</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>435</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcfpherd</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>435</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcfrlb</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>476</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcfrrb</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>476</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vczb</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>447</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vczid</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>449</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vczdrn</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>449</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vclzh</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>447</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vclzx</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>453</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vclzxlb</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>448</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vclzw</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>429</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcmplexp</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>430</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcmplexq</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>388</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcmplexr</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>392</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcmplexw</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>388</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcmqexpf</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>430</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcmqexpq</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>431</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcmqexpb</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>393</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcmqexpf</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>396</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcmqexpf</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>394</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcmqexpq</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>397</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcmqexpw</td>
<td>i</td>
<td>i</td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>395</td>
<td></td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 14 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcmpglub[ ]</td>
<td>Vector Compare Greater Than Unsigned Byte VC-form</td>
</tr>
<tr>
<td>vcmpglud[ ]</td>
<td>Vector Compare Greater Than Unsigned Doubleword VC-form</td>
</tr>
<tr>
<td>vcmpgluh[ ]</td>
<td>Vector Compare Greater Than Unsigned Halfword VC-form</td>
</tr>
<tr>
<td>vcmpglul[ ]</td>
<td>Vector Compare Greater Than Unsigned Quadword VC-form</td>
</tr>
<tr>
<td>vcmpneb[ ]</td>
<td>Vector Compare Not Equal Byte VC-form</td>
</tr>
<tr>
<td>vcmpnew[ ]</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vcmpnewv[ ]</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vcmpsq</td>
<td>Vector Compare Signed Quadword VX-form</td>
</tr>
<tr>
<td>vcmpsuq</td>
<td>Vector Compare Unsigned Quadword VX-form</td>
</tr>
<tr>
<td>vctzw</td>
<td>Vector Count Trailing Zeros Word AVX-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Count Trailing Zeros Halfword VX-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Count Trailing Zeros Least Significant Bits Byte VX-form</td>
</tr>
<tr>
<td>vctzw</td>
<td>Vector Count Trailing Zeros Word VX-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Count Trailing Zeros Halfword VX-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Count Trailing Zero Least Significant Bits Byte VX-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Count Trailing Zeros Word VX-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Count Trailing Zero Least Significant Bits Byte VX-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Count Trailing Zeros Word VX-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Count Trailing Zero Least Significant Bits Byte VX-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Count Trailing Zeros Word VX-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Count Trailing Zero Least Significant Bits Byte VX-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>vctzwh</td>
<td>Vector Compare Not Equal Word VC-form</td>
</tr>
<tr>
<td>vctzwsb</td>
<td>Vector Compare Not Equal Halfword VC-form</td>
</tr>
<tr>
<td>Instruction</td>
<td>Book</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 16 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vinsd</code></td>
<td></td>
<td></td>
<td></td>
<td>vinsd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>319</td>
<td>Vector Insert Doubleword from GPR using immediate-specified index VX-form</td>
</tr>
<tr>
<td><code>vinsdx</code></td>
<td></td>
<td></td>
<td></td>
<td>vinsdx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>318</td>
<td>Vector Insert Doubleword from GPR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td><code>vinserter</code></td>
<td></td>
<td></td>
<td></td>
<td>vinserter</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>314</td>
<td>Vector Insert Doubleword from GPR using GPR-specified Right-Index VX-form</td>
</tr>
<tr>
<td><code>vinserterh</code></td>
<td></td>
<td></td>
<td></td>
<td>vinserterh</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>313</td>
<td>Vector Insert Doubleword from GPR using GPR-specified Right-Index VX-form</td>
</tr>
<tr>
<td><code>vinsertb</code></td>
<td></td>
<td></td>
<td></td>
<td>vinsertb</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>314</td>
<td>Vector Insert Doubleword from GPR using immediate-specified index VX-form</td>
</tr>
<tr>
<td><code>vinsertd</code></td>
<td></td>
<td></td>
<td></td>
<td>vinsertd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>314</td>
<td>Vector Insert Doubleword from GPR using immediate-specified index VX-form</td>
</tr>
<tr>
<td><code>vinshlx</code></td>
<td></td>
<td></td>
<td></td>
<td>vinshlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>316</td>
<td>Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td><code>vinshrx</code></td>
<td></td>
<td></td>
<td></td>
<td>vinshrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>316</td>
<td>Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td><code>vinshvlx</code></td>
<td></td>
<td></td>
<td></td>
<td>vinshvlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>321</td>
<td>Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td><code>vinshvrx</code></td>
<td></td>
<td></td>
<td></td>
<td>vinshvrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>321</td>
<td>Vector Insert Halfword from GPR using GPR-specified Right-Index VX-form</td>
</tr>
<tr>
<td><code>vinsw</code></td>
<td></td>
<td></td>
<td></td>
<td>vinsw</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>319</td>
<td>Vector Insert Halfword from GPR using immediate-specified index VX-form</td>
</tr>
<tr>
<td><code>vinswlx</code></td>
<td></td>
<td></td>
<td></td>
<td>vinswlx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>317</td>
<td>Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td><code>vinswrx</code></td>
<td></td>
<td></td>
<td></td>
<td>vinswrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>317</td>
<td>Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form</td>
</tr>
<tr>
<td><code>vinswvrx</code></td>
<td></td>
<td></td>
<td></td>
<td>vinswvrx</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>322</td>
<td>Vector Insert Halfword from GPR using GPR-specified Right-Index VX-form</td>
</tr>
<tr>
<td><code>vlogelp</code></td>
<td></td>
<td></td>
<td></td>
<td>vlogelp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>433</td>
<td>Vector Log Base 2 Estimate Floating-Point VX-form</td>
</tr>
<tr>
<td><code>vmaaddfp</code></td>
<td></td>
<td></td>
<td></td>
<td>vmaaddfp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>423</td>
<td>Vector Multiply-Add Floating-Point VX-form</td>
</tr>
<tr>
<td><code>vmaaddp</code></td>
<td></td>
<td></td>
<td></td>
<td>vmaaddp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>424</td>
<td>Vector Multiply-Add Floating-Point VX-form</td>
</tr>
<tr>
<td><code>vmaxsb</code></td>
<td></td>
<td></td>
<td></td>
<td>vmaxsb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>380</td>
<td>Vector Maximum Signed Byte VX-form</td>
</tr>
<tr>
<td><code>vmaxsd</code></td>
<td></td>
<td></td>
<td></td>
<td>vmaxsd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>381</td>
<td>Vector Maximum Signed Doubleword VX-form</td>
</tr>
<tr>
<td><code>vmaxsh</code></td>
<td></td>
<td></td>
<td></td>
<td>vmaxsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>381</td>
<td>Vector Maximum Signed Halfword VX-form</td>
</tr>
<tr>
<td><code>vmaxsw</code></td>
<td></td>
<td></td>
<td></td>
<td>vmaxsw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>382</td>
<td>Vector Maximum Signed Word VX-form</td>
</tr>
<tr>
<td><code>vmaxub</code></td>
<td></td>
<td></td>
<td></td>
<td>vmaxub</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>380</td>
<td>Vector Maximum Signed Byte VX-form</td>
</tr>
<tr>
<td><code>vmaxud</code></td>
<td></td>
<td></td>
<td></td>
<td>vmaxud</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>383</td>
<td>Vector Maximum Signed Doubleword VX-form</td>
</tr>
<tr>
<td><code>vmaxuh</code></td>
<td></td>
<td></td>
<td></td>
<td>vmaxuh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>381</td>
<td>Vector Maximum Signed Halfword VX-form</td>
</tr>
<tr>
<td><code>vmaxuw</code></td>
<td></td>
<td></td>
<td></td>
<td>vmaxuw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>382</td>
<td>Vector Maximum Signed Word VX-form</td>
</tr>
<tr>
<td><code>vmhaddhshs</code></td>
<td></td>
<td></td>
<td></td>
<td>vmhaddhshs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>351</td>
<td>Vector Multiply-High-Add Signed Halfword Saturate VX-form</td>
</tr>
<tr>
<td><code>vmhraddshs</code></td>
<td></td>
<td></td>
<td></td>
<td>vmhraddhsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>351</td>
<td>Vector Multiply-High-Add Signed Halfword Saturate VX-form</td>
</tr>
<tr>
<td><code>vmnlp</code></td>
<td></td>
<td></td>
<td></td>
<td>vmlp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>424</td>
<td>Vector Minimum Floating-Point VX-form</td>
</tr>
<tr>
<td><code>vmnsb</code></td>
<td></td>
<td></td>
<td></td>
<td>vmsb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>384</td>
<td>Vector Minimum Signed Byte VX-form</td>
</tr>
<tr>
<td><code>vmnsd</code></td>
<td></td>
<td></td>
<td></td>
<td>vmsd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>387</td>
<td>Vector Minimum Signed Doubleword VX-form</td>
</tr>
<tr>
<td><code>vmnsh</code></td>
<td></td>
<td></td>
<td></td>
<td>vmsnh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>385</td>
<td>Vector Minimum Signed Halfword VX-form</td>
</tr>
<tr>
<td><code>vmnsiw</code></td>
<td></td>
<td></td>
<td></td>
<td>vmsiw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>386</td>
<td>Vector Minimum Signed Word VX-form</td>
</tr>
<tr>
<td><code>vmnub</code></td>
<td></td>
<td></td>
<td></td>
<td>vmnub</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>384</td>
<td>Vector Minimum Signed Byte VX-form</td>
</tr>
<tr>
<td><code>vmnud</code></td>
<td></td>
<td></td>
<td></td>
<td>vmnud</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>387</td>
<td>Vector Minimum Signed Doubleword VX-form</td>
</tr>
<tr>
<td><code>vmnuh</code></td>
<td></td>
<td></td>
<td></td>
<td>vmnuh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>385</td>
<td>Vector Minimum Signed Halfword VX-form</td>
</tr>
<tr>
<td><code>vmnuw</code></td>
<td></td>
<td></td>
<td></td>
<td>vmnuw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>386</td>
<td>Vector Minimum Signed Word VX-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 17 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets 2</th>
<th>Linux Optional Category 2</th>
<th>Always Optional Category 4</th>
<th>Mnemonic</th>
<th>Version 5</th>
<th>Privilege 6</th>
<th>Mode Dep 7</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmladuhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>352</td>
<td>Vector Multiply-Low-Add Unsigned Halfword Modulo VX-form</td>
</tr>
<tr>
<td>vmodaluhm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>365</td>
<td>Vector Modulo Signed Doubleword VX-form</td>
</tr>
<tr>
<td>vmodalq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>366</td>
<td>Vector Modulo Signed Quadword VX-form</td>
</tr>
<tr>
<td>vmodalw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>364</td>
<td>Vector Modulo Signed Word VX-form</td>
</tr>
<tr>
<td>vmodald</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>365</td>
<td>Vector Modulo Signed Doubleword VX-form</td>
</tr>
<tr>
<td>vmodalq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>366</td>
<td>Vector Modulo Signed Quadword VX-form</td>
</tr>
<tr>
<td>vmodalw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>364</td>
<td>Vector Modulo Signed Word VX-form</td>
</tr>
<tr>
<td>vmsgnew</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>292</td>
<td>Vector Merge Even Word VX-form</td>
</tr>
<tr>
<td>vmsgnhb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge High Byte VX-form</td>
</tr>
<tr>
<td>vmsgnhw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>290</td>
<td>Vector Merge High Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge Low Byte VX-form</td>
</tr>
<tr>
<td>vmsgnh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>290</td>
<td>Vector Merge High Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>291</td>
<td>Vector Merge High Word VX-form</td>
</tr>
<tr>
<td>vmsgnh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge Low Byte VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>290</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>291</td>
<td>Vector Merge Low Word VX-form</td>
</tr>
<tr>
<td>vmsgnh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>290</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>291</td>
<td>Vector Merge Low Word VX-form</td>
</tr>
<tr>
<td>vmsgnh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>290</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>291</td>
<td>Vector Merge Low Word VX-form</td>
</tr>
<tr>
<td>vmsgnh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>290</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>291</td>
<td>Vector Merge Low Word VX-form</td>
</tr>
<tr>
<td>vmsgnh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>290</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>291</td>
<td>Vector Merge Low Word VX-form</td>
</tr>
<tr>
<td>vmsgnh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>290</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>291</td>
<td>Vector Merge Low Word VX-form</td>
</tr>
<tr>
<td>vmsgnh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>289</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>290</td>
<td>Vector Merge Low Halfword VX-form</td>
</tr>
<tr>
<td>vmsgnw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>291</td>
<td>Vector Merge Low Word VX-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 18 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmuuh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>342</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmuuw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>344</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vmuuw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>347</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnand</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>403</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vncipher</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>436</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnCipher</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>436</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnegd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>371</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnegw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>371</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnmsublp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>423</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnor</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>403</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vor</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>403</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vorc</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>403</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpdpd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>454</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vperm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>296</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpermr</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>296</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpermxor</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>444</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpxtd</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>455</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpkpx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>276</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpsds</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>281</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpsdus</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>281</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vphss</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>279</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vphsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>279</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vphsw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>280</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vphsw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>280</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpkdum</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpkdus</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vphum</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>282</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vphus</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>282</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vphw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vphuw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpmusmb</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>440</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpmusmd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>443</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpmusmh</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>441</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpmusmw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>442</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpopcntb</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>457</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vpopcntl</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>456</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vprtydb</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>459</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vprtydbw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>459</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsmp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>434</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrfim</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>427</td>
<td></td>
</tr>
<tr>
<td>vx100 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vrfin</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>427</td>
<td></td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 19 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrfip</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Round to Floating-Point Integer toward +Infinity VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrfiz</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Round to Floating-Point Integer toward Zero VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Byte VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrld</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Doubleword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrldmi</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Doubleword then Mask Insert VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrdnm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Doubleword then AND with Mask VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrrh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Halfword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrrq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Quadword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrlqmq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Quadword then Mask Insert VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrlqnm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Quadword then AND with Mask VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrlw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Word VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrlwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Word then Mask Insert VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vrlwmm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>Vector Rotate Left Word then AND with Mask VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsqtrefp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Reciprocal Square Root Estimate Floating-Point VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsbox</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Vector AES SubBytes VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vaal</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Select VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsasigmz</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Vector SHA-512 Sigma Doubleword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsasigmac</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Vector SHA-256 Sigma Word VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Left VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Left Byte VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Left Doubleword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsld</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Left Doubleword then Mask Insert VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vslh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Left Halfword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vshl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Left by Octet VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsaq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Left Quadword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsal</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Left Variable VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsalw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Left Word VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vspltb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Splat Byte VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vspltbh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Splat Halfword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vspltlsb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Splat Immediate Signed Byte VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vspltlish</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Splat Immediate Signed Halfword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vspltlsw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Splat Immediate Signed Word VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vspltlh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Splat Word VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsr</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsrab</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Algebraic Byte VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsrad</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Algebraic Doubleword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsrah</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Algebraic Halfword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsraq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Algebraic Quadword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsraw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Algebraic Word VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsrb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Byte VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vard</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Doubleword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>verdi</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Doubleword by Octet Immediate VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>varh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Halfword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsro</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right by Octet VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsrq</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Quadword VX-form</td>
</tr>
<tr>
<td>001111</td>
<td>111110 011100 001110</td>
<td>I</td>
<td>I</td>
<td>vsrv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>Vector Shift Right Variable VX-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 20 of 30)

1500 Power ISA™ Appendices
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Def</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>417</td>
<td>Vector Shift Right Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vstribh</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>474</td>
<td>Vector String Isolate Byte Left-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vstrih[.]</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>475</td>
<td>Vector String Isolate Byte Right-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vstrih[.]</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>475</td>
<td>Vector String Isolate Halfword Left-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vstrih[.]</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>474</td>
<td>Vector String Isolate Halfword Right-justified VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubc  uq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>338</td>
<td>Vector Subtract &amp; write Carry-out Unsigned Quadword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsubc  uw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>331</td>
<td>Vector Subtract &amp; Write Carry-out Unsigned Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsbecuteq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>338</td>
<td>Vector Subtract Extended &amp; write Carry-out Unsigned Quadword VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsbecuteq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>337</td>
<td>Vector Subtract Extended Unsign  ed Quadword Modulo VA-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>422</td>
<td>Vector Subtract Floating-Point VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>331</td>
<td>Vector Subtract Signed Byte Sat  urate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>332</td>
<td>Vector Subtract Signed Halfword Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>332</td>
<td>Vector Subtract Signed Word Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>333</td>
<td>Vector Subtract Unsigned Byte Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>335</td>
<td>Vector Subtract Unsigned Byte Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>334</td>
<td>Vector Subtract Unsigned Doubleword Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>333</td>
<td>Vector Subtract Unsigned Halfword Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>335</td>
<td>Vector Subtract Unsigned Halfword Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>336</td>
<td>Vector Subtract Unsigned Word Modulo VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>334</td>
<td>Vector Subtract Unsigned Word Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>336</td>
<td>Vector Subtract Unsigned Word Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>336</td>
<td>Vector Subtract Unsigned Word Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>339</td>
<td>Vector Subtract Unsigned Quarter Signed Byte Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>370</td>
<td>Vector Subtract Unsigned Quarter Signed Byte Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>367</td>
<td>Vector Subtract Signed Word Saturate VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Subtract Pack High Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Subtract Pack High Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Subtract Pack High Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>286</td>
<td>Vector Subtract Pack Halfword PX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>285</td>
<td>Vector Subtract Pack Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>285</td>
<td>Vector Subtract Pack Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>285</td>
<td>Vector Subtract Pack Signed Halfword VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>287</td>
<td>Vector Subtract Pack Low Signed PX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>287</td>
<td>Vector Subtract Pack Low Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>287</td>
<td>Vector Subtract Pack Low Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Subtract Unpack Low Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>369</td>
<td>Vector Subtract Unpack Low Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vsin  ds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>369</td>
<td>Vector Subtract Unpack Low Signed Word VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vxor</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>403</td>
<td>Vector Logical XOR VX-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>1080</td>
<td>Wait X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xor[.]</td>
<td>P1</td>
<td></td>
<td></td>
<td>100</td>
<td>XOR X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xori</td>
<td>P1</td>
<td></td>
<td></td>
<td>100</td>
<td>XOR Immediate D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xoris</td>
<td>P1</td>
<td></td>
<td></td>
<td>100</td>
<td>XOR Immediate Shifted D-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xabsdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>658</td>
<td>VSX Scalar Absolute Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xabsdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>658</td>
<td>VSX Scalar Absolute Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xadddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>659</td>
<td>VSX Scalar Add Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xadddp[o]v3.0</td>
<td>666</td>
<td>VSX Scalar Add Quad-Precision (using round to Odd) X-form</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xaddsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>664</td>
<td>VSX Scalar Add Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xaddsp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>670</td>
<td>VSX Scalar Compare Equal Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcmppqdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>671</td>
<td>VSX Scalar Compare Equal Double-Precision X-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 21 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Instruction</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscmpexpdp</td>
<td>v3.0</td>
<td>VSX Scalar Compare Exponents Double-Precision XX3-form</td>
<td></td>
<td></td>
<td>668</td>
<td></td>
</tr>
<tr>
<td>xscmpexpqp</td>
<td>v3.0</td>
<td>VSX Scalar Compare Exponents Quad-Precision X-form</td>
<td></td>
<td></td>
<td>669</td>
<td></td>
</tr>
<tr>
<td>xscmpgedp</td>
<td>v3.0</td>
<td>VSX Scalar Compare Greater Than or Equal Double-Precision XX3-form</td>
<td></td>
<td></td>
<td>672</td>
<td></td>
</tr>
<tr>
<td>xscmpgeqp</td>
<td>v3.1</td>
<td>VSX Scalar Compare Greater Than or Equal Quad-Precision X-form</td>
<td></td>
<td></td>
<td>673</td>
<td></td>
</tr>
<tr>
<td>xscmpgtdp</td>
<td>v3.0</td>
<td>VSX Scalar Compare Greater Than Double-Precision XX3-form</td>
<td></td>
<td></td>
<td>674</td>
<td></td>
</tr>
<tr>
<td>xscmpgtqp</td>
<td>v3.1</td>
<td>VSX Scalar Compare Greater Than Quad-Precision X-form</td>
<td></td>
<td></td>
<td>675</td>
<td></td>
</tr>
<tr>
<td>xscmpodp</td>
<td>v2.06</td>
<td>VSX Scalar Compare Ordered Double-Precision XX3-form</td>
<td></td>
<td></td>
<td>676</td>
<td></td>
</tr>
<tr>
<td>xscmpoqp</td>
<td>v2.06</td>
<td>VSX Scalar Compare Ordered Quad-Precision X-form</td>
<td></td>
<td></td>
<td>678</td>
<td></td>
</tr>
<tr>
<td>xscmpudp</td>
<td>v2.06</td>
<td>VSX Scalar Compare Unordered Double-Precision XX3-form</td>
<td></td>
<td></td>
<td>679</td>
<td></td>
</tr>
<tr>
<td>xscmpuqp</td>
<td>v2.06</td>
<td>VSX Scalar Compare Unordered Quad-Precision X-form</td>
<td></td>
<td></td>
<td>680</td>
<td></td>
</tr>
<tr>
<td>xscvdphp</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round Double-Precision to Half-Precision format XX2-form</td>
<td></td>
<td></td>
<td>683</td>
<td></td>
</tr>
<tr>
<td>xscvdqpp</td>
<td>v3.0</td>
<td>VSX Scalar Convert Double-Precision to Quad-Precision format X-form</td>
<td></td>
<td></td>
<td>684</td>
<td></td>
</tr>
<tr>
<td>xscvdsp</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round Double-Precision to Single-Precision format XX2-form</td>
<td></td>
<td></td>
<td>685</td>
<td></td>
</tr>
<tr>
<td>xscvdspn</td>
<td>v2.07</td>
<td>VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form</td>
<td></td>
<td></td>
<td>686</td>
<td></td>
</tr>
<tr>
<td>xscvdxds</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Doubledword format XX2-form</td>
<td></td>
<td></td>
<td>687</td>
<td></td>
</tr>
<tr>
<td>xscvdxsds</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Doubledword format XX2-form</td>
<td></td>
<td></td>
<td>689</td>
<td></td>
</tr>
<tr>
<td>xscvdxsxs</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Word format XX2-form</td>
<td></td>
<td></td>
<td>691</td>
<td></td>
</tr>
<tr>
<td>xscvdxuds</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubledword format XX2-form</td>
<td></td>
<td></td>
<td>691</td>
<td></td>
</tr>
<tr>
<td>xscvdxxds</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format XX2-form</td>
<td></td>
<td></td>
<td>693</td>
<td></td>
</tr>
<tr>
<td>xscvhdps</td>
<td>v3.0</td>
<td>VSX Scalar Convert Half-Precision to Double-Precision format XX2-form</td>
<td></td>
<td></td>
<td>695</td>
<td></td>
</tr>
<tr>
<td>xscvgdpd</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] X-form</td>
<td></td>
<td></td>
<td>696</td>
<td></td>
</tr>
<tr>
<td>xscvgpsdz</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Doubledword format X-form</td>
<td></td>
<td></td>
<td>697</td>
<td></td>
</tr>
<tr>
<td>xscvgpsqz</td>
<td>v3.1</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Quadrword X-form</td>
<td></td>
<td></td>
<td>699</td>
<td></td>
</tr>
<tr>
<td>xscvgpswz</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
<td></td>
<td></td>
<td>701</td>
<td></td>
</tr>
<tr>
<td>xscvgpsdz</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
<td></td>
<td></td>
<td>701</td>
<td></td>
</tr>
<tr>
<td>xscvgpuqz</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Quadrword X-form</td>
<td></td>
<td></td>
<td>705</td>
<td></td>
</tr>
<tr>
<td>xscvgwuwz</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form</td>
<td></td>
<td></td>
<td>707</td>
<td></td>
</tr>
<tr>
<td>xscvsdpd</td>
<td>v3.0</td>
<td>VSX Scalar Convert Signed Doubledword to Quad-Precision format X-form</td>
<td></td>
<td></td>
<td>714</td>
<td></td>
</tr>
<tr>
<td>xscvudp</td>
<td>v2.06</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format XX2-form</td>
<td></td>
<td></td>
<td>709</td>
<td></td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 22 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xscvstdp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>710</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xscvqdp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>711</td>
<td>VSX Scalar Convert with round Signed Quadword to Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xscvsxddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>712</td>
<td>VSX Scalar Convert with round Signed Doubleword to Double-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xscvsxsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>713</td>
<td>VSX Scalar Convert with round Signed Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xscvudp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>714</td>
<td>VSX Scalar Convert Unsigned Doubleword to Quad-Precision format X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xscvuuqp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>715</td>
<td>VSX Scalar Convert with round Unsigned Quadword to Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xscvxudp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>716</td>
<td>VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xscvxsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>717</td>
<td>VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsdvdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>718</td>
<td>VSX Scalar Divide Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsdivdpl[0]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>719</td>
<td>VSX Scalar Divide Quad-Precision [using round to Odd] X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsdivsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>721</td>
<td>VSX Scalar Divide Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxieixdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>723</td>
<td>VSX Scalar Insert Exponent Double-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxieixsp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>724</td>
<td>VSX Scalar Insert Exponent Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmadddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>725</td>
<td>VSX Scalar Multiply-Add Type-A Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaddasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>726</td>
<td>VSX Scalar Multiply-Add Type-A Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaddmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>727</td>
<td>VSX Scalar Multiply-Add Type-M Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaddmsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>728</td>
<td>VSX Scalar Multiply-Add Type-M Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaddop[0]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>731</td>
<td>VSX Scalar Multiply-Add Quad-Precision [using round to Odd] X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmacxdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>736</td>
<td>VSX Scalar Maximum Type-C Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmacxp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>738</td>
<td>VSX Scalar Maximum Type-C Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaxdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>739</td>
<td>VSX Scalar Maximum Type-J Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmaxjdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>743</td>
<td>VSX Scalar Minimum Type-C Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmincdpx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>745</td>
<td>VSX Scalar Minimum Type-C Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmincap</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>741</td>
<td>VSX Scalar Minimum Type-C Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmindp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>746</td>
<td>VSX Scalar Minimum Type-J Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmindjdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>747</td>
<td>VSX Scalar Minimum Type-A Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmsubdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>748</td>
<td>VSX Scalar Multiply-Subtract Type-A Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmsubasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>751</td>
<td>VSX Scalar Multiply-Subtract Type-A Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmsubmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>749</td>
<td>VSX Scalar Multiply-Subtract Type-M Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmsubmsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>751</td>
<td>VSX Scalar Multiply-Subtract Type-M Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmsubop[0]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>754</td>
<td>VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd] X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmulp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>757</td>
<td>VSX Scalar Multiply Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsmulop[0]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>759</td>
<td>VSX Scalar Multiply Quad-Precision [using round to Odd] X-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 23 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Depol</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmulsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>761</td>
<td>VSX Scalar Multiply Single-Precision XX3-form</td>
</tr>
<tr>
<td>xsnabsdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>763</td>
<td>VSX Scalar Negative Absolute Double-Precision XX2-form</td>
</tr>
<tr>
<td>xsnabsdqp</td>
<td></td>
<td>BFP128</td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>763</td>
<td>VSX Scalar Negative Absolute Quad-Precision X-form</td>
</tr>
<tr>
<td>xsnegdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>764</td>
<td>VSX Scalar Negate Double-Precision XX2-form</td>
</tr>
<tr>
<td>xsnegdqp</td>
<td></td>
<td>BFP128</td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>764</td>
<td>VSX Scalar Negate Quad-Precision X-form</td>
</tr>
<tr>
<td>xsnmaddadp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>765</td>
<td>VSX Scalar Negative Multiply-Add Type-A Double-Precision XX2-form</td>
</tr>
<tr>
<td>xsnmaddasp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>770</td>
<td>VSX Scalar Negative Multiply-Add Type-A Single-Precision XX2-form</td>
</tr>
<tr>
<td>xsnmaddmdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>775</td>
<td>VSX Scalar Negative Multiply-Add Type-M Double-Precision XX2-form</td>
</tr>
<tr>
<td>xsnmaddmsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>770</td>
<td>VSX Scalar Negative Multiply-Add Type-M Single-Precision XX2-form</td>
</tr>
<tr>
<td>xsnmaddmq[e]</td>
<td></td>
<td>BFP128</td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>773</td>
<td>VSX Scalar Negative Multiply-Add Quad-Precision using round to Odd X-form</td>
</tr>
<tr>
<td>xsnmsubadp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>776</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Double-Precision XX2-form</td>
</tr>
<tr>
<td>xsnmsubasdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>779</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Single-Precision XX2-form</td>
</tr>
<tr>
<td>xsnmsubmdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>776</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Double-Precision XX2-form</td>
</tr>
<tr>
<td>xsnmsubmdsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>779</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Single-Precision XX2-form</td>
</tr>
<tr>
<td>xsnmsubmq[e]</td>
<td></td>
<td>BFP128</td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>782</td>
<td>VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd] X-form</td>
</tr>
<tr>
<td>xsrdpi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>785</td>
<td>VSX Scalar Round to Double-Precision Integer using round to Nearest Away XX2-form</td>
</tr>
<tr>
<td>xsrdpic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>786</td>
<td>VSX Scalar Round to Double-Precision Integer exact using Current rounding mode XX2-form</td>
</tr>
<tr>
<td>xsrdpim</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>787</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward -Infinity XX2-form</td>
</tr>
<tr>
<td>xsrdpip</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>788</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form</td>
</tr>
<tr>
<td>xsrdpiz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>789</td>
<td>VSX Scalar Round to Double-Precision Integer using round toward Zero XX2-form</td>
</tr>
<tr>
<td>xssredp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>790</td>
<td>VSX Scalar Reciprocal Estimate Double-Precision XX2-form</td>
</tr>
<tr>
<td>xssresp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>791</td>
<td>VSX Scalar Reciprocal Estimate Single-Precision XX2-form</td>
</tr>
<tr>
<td>xssrqi[x]</td>
<td></td>
<td>BFP128</td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>793</td>
<td>VSX Scalar Round to Quad-Precision Integer [with Inexact] XX3-form</td>
</tr>
<tr>
<td>xssrgp[x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>795</td>
<td>VSX Scalar Round Quadr-Precision to Double-Extended Precision XX2-form</td>
</tr>
<tr>
<td>xssrp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>797</td>
<td>VSX Scalar Round to Single-Precision XX2-form</td>
</tr>
<tr>
<td>xssrgtedp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>798</td>
<td>VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form</td>
</tr>
<tr>
<td>xssrgtesp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>799</td>
<td>VSX Scalar Reciprocal Square Root Estimate Single-Precision XX2-form</td>
</tr>
<tr>
<td>xssqrtdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>801</td>
<td>VSX Scalar Square Root Double-Precision XX2-form</td>
</tr>
<tr>
<td>xssqrtdp[c]</td>
<td></td>
<td>BFP128</td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>803</td>
<td>VSX Scalar Square Root Quad-Precision using round to Odd X-form</td>
</tr>
<tr>
<td>xssqrtsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>805</td>
<td>VSX Scalar Square Root Single-Precision XX2-form</td>
</tr>
<tr>
<td>xssubdp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>807</td>
<td>VSX Scalar Subtract Double-Precision XX2-form</td>
</tr>
<tr>
<td>xssubdp[c]</td>
<td></td>
<td>BFP128</td>
<td></td>
<td></td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>809</td>
<td>VSX Scalar Subtract Quad-Precision [using round to Odd] X-form</td>
</tr>
<tr>
<td>xssubsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
<td>811</td>
<td>VSX Scalar Subtract Single-Precision XX2-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 24 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xsvdivdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Scalar test for software Divide Double-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xstsqrtdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Scalar test for software Square Root Double-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xststdcdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VEX Scalar test Data Class Double-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xststdcsp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VEX Scalar test Data Class Single-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xssexpdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VEX Scalar Extract Exponent Double-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xssexpqp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VEX Scalar Extract Exponent Quad-Precision X-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xststdcdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VEX Scalar Extract Significand Double-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xststdcsp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
<td>VEX Scalar Extract Significand Single-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvbabsdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Absolute Value Double-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvbabsqp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Absolute Value Single-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvdaddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Add Double-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvdadsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Add Single-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xbf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector bisadd16 GER (Rank-2 Update) X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xbf16ger2mp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector bisadd16 GER (Rank-2 Update) Negative multiply, Negative accumulate X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xbf16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector bisadd16 GER (Rank-2 Update) Positive multiply, Negative accumulate X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xbf16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector bisadd16 GER (Rank-2 Update) Positive multiply, Positive accumulate X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcmsigdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Compare Equal To Double-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcmpeqdp[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Compare Equal To Single-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcmpeqsp[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Compare Greater Than or Equal To Single-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcmpgedp[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Compare Greater Than or Equal To Double-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcmpgesp[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Compare Greater Than or Equal To Single-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcmpgsp[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Compare Greater Than Double-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcpsigndp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Copy Sign Double-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcpsignsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Copy Sign Single-Precision X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvf16sp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Convert bisadd16 to Single-Precision format X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvdsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Convert with round Double-Precision to Single-Precision format X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcdpsxsds</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Convert with round to Zero Double-Precision to Signed Doubleword format X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcdpsxs</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Convert with round to Double-Precision to Signed Word format X2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcvdpsxsds</td>
<td>v2.06</td>
<td></td>
<td></td>
<td></td>
<td>VEX Vector Convert with round to Zero Double-Precision to Signed Doubleword format X2-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 25 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets³</th>
<th>Linux Optional Category²</th>
<th>Always Optional Category²</th>
<th>Mnemonic</th>
<th>Version⁵</th>
<th>Privilege⁶</th>
<th>Mode Dep⁷</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvdpuxw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>847</td>
<td>VSX Vector Convert with round to zero Double-Precision to Unsigned Word format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvchp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>849</td>
<td>VSX Vector Convert Half-Precision to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvspbf16</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>850</td>
<td>VSX Vector Convert with round Single-Precision to bfloat16 format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvcsd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>853</td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvpsdxw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>855</td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Word format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvtx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>857</td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvtxpp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>859</td>
<td>VSX Vector Convert with round to zero Single-Precision to Unsigned Word format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvtxpp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>861</td>
<td>VSX Vector Convert with round Signed Doubleword to Double-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvtxppd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>862</td>
<td>VSX Vector Convert with round Signed Doubleword to Double-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvtpux</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>863</td>
<td>VSX Vector Convert Signed Word to Double-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvtpw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>863</td>
<td>VSX Vector Convert with round Signed Word to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvtdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>864</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Double-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvudpp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>865</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvudsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>866</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xcvwsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>866</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxdivdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>867</td>
<td>VSX Vector Divide Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxdivsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>869</td>
<td>VSX Vector Divide Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvf16ger2</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>VSX Vector Convert 16-bit Floating-Point GER (rank-2 update) XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvf16ger2nm</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>VSX Vector Convert 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvf16ger2np</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>VSX Vector Convert 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvf16ger2pn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>VSX Vector Convert 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvf16ger2pp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>871</td>
<td>VSX Vector Convert 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvf32ger</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>VSX Vector Convert 32-bit Floating-Point GER (rank-1 update) XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvf32genn</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>VSX Vector Convert 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvf32gemp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>VSX Vector Convert 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvf32gerp</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>VSX Vector Convert 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvf32gerr</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>875</td>
<td>VSX Vector Convert 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 26 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Back</th>
<th>Compliant Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111001</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi64ger</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi64geen</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi64gemp</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi64gerpn</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi64gerpp</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>879</td>
<td>VSX Vector 64-bit Floating-Point GER (rank-1 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi16ger2</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>891</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi16ger2pp</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>891</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi16ger2s</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>893</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi16ger2pp</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>893</td>
<td>VSX Vector 16-bit Signed Integer GER (rank-2 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi4ger8</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>883</td>
<td>VSX Vector 4-bit Signed Integer GER (rank-8 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi4ger8pp</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>883</td>
<td>VSX Vector 4-bit Signed Integer GER (rank-8 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi6ger4</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>886</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi8ger4pp</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>886</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update)</td>
</tr>
<tr>
<td>111011</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvi8ger4pp</td>
<td>v3.1</td>
<td>...</td>
<td>...</td>
<td>889</td>
<td>VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update)</td>
</tr>
<tr>
<td>111010</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xviexpdp</td>
<td>v3.0</td>
<td>...</td>
<td>...</td>
<td>896</td>
<td>VSX Vector Insert Exponent Double-Precision XX3-form</td>
</tr>
<tr>
<td>111010</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xviexpsp</td>
<td>v3.0</td>
<td>...</td>
<td>...</td>
<td>896</td>
<td>VSX Vector Insert Exponent Single-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmaddadp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>897</td>
<td>VSX Vector Multiply-Add Type-A Double-Precision XX2-form</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmaddasp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>900</td>
<td>VSX Vector Multiply-Add Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmaddmdp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>897</td>
<td>VSX Vector Multiply-Add Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmaddmsp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>900</td>
<td>VSX Vector Multiply-Add Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmaddsp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>903</td>
<td>VSX Vector Maximum Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmindp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>905</td>
<td>VSX Vector Maximum Single-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmindsp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>907</td>
<td>VSX Vector Minimum Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmindsp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>909</td>
<td>VSX Vector Minimum Single-Precision XX2-form</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmsubadp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>911</td>
<td>VSX Vector Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmsubasp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>914</td>
<td>VSX Vector Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmsubmdp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>911</td>
<td>VSX Vector Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmsubmsp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>914</td>
<td>VSX Vector Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmuldp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>917</td>
<td>VSX Vector Multiply Double-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xvmulsdp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>919</td>
<td>VSX Vector Multiply Single-Precision XX3-form</td>
</tr>
<tr>
<td>111100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>xnabsdp</td>
<td>v2.06</td>
<td>...</td>
<td>...</td>
<td>921</td>
<td>VSX Vector Negative Absolute Double-Precision XX2-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 27 of 30)
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Book</th>
<th>Compliancy Subsets²</th>
<th>Linux Optional Category³</th>
<th>Always Optional Category⁴</th>
<th>Mnemonic</th>
<th>Version⁵</th>
<th>Privilege⁶</th>
<th>Mode Dep⁷</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnabssp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>921</td>
<td>VSX Vector Negative Absolute Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnegdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>922</td>
<td>VSX Vector Negate Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnegisp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>922</td>
<td>VSX Vector Negate Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnaddsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>923</td>
<td>VSX Vector Negative Multiply-Add Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnaddisp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>927</td>
<td>VSX Vector Negative Multiply-Add Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnaddmp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>923</td>
<td>VSX Vector Negative Multiply-Add Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnaddmisp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>927</td>
<td>VSX Vector Negative Multiply-Add Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnmsubadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>930</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnmsubasp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>933</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnmsubmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>930</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxnmsubmisp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>933</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrdpi</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>936</td>
<td>VSX Vector Round to Double-Precision Integer using round to Nearest Away XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrdpic</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>937</td>
<td>VSX Vector Round to Double-Precision Integer Exact using Current rounding mode XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrdpm</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>938</td>
<td>VSX Vector Round to Double-Precision Integer using round toward -Infinity XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrdpp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>939</td>
<td>VSX Vector Round to Double-Precision Integer using round toward +Infinity XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrdpq</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>939</td>
<td>VSX Vector Round to Double-Precision Integer using round toward Zero XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxredp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>940</td>
<td>VSX Vector Reciprocal Estimate Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxresp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>941</td>
<td>VSX Vector Reciprocal Estimate Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrspl</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>942</td>
<td>VSX Vector Round to Single-Precision Integer using round to Nearest Away XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrsptic</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>943</td>
<td>VSX Vector Round to Single-Precision Integer Exact using Current rounding mode XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrsppm</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>944</td>
<td>VSX Vector Round to Single-Precision Integer using round toward +Infinity XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrspp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>945</td>
<td>VSX Vector Round to Single-Precision Integer using round toward -Infinity XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrsps</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>945</td>
<td>VSX Vector Round to Single-Precision Integer using round toward Zero XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrsqrtedp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>946</td>
<td>VSX Vector Reciprocal Square Root Estimate Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxrsqtesp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>947</td>
<td>VSX Vector Reciprocal Square Root Estimate Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsqrtdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>948</td>
<td>VSX Vector Square Root Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsqrtsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>949</td>
<td>VSX Vector Square Root Single-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsubdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>950</td>
<td>VSX Vector Subtract Double-Precision XX2-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xxsubsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>952</td>
<td>VSX Vector Subtract Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvtddvdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>954</td>
<td>VSX Vector Test for software Divide Double-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvtdivsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>955</td>
<td>VSX Vector Test for software Divide Single-Precision XX3-form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xvtflsb</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>959</td>
<td>VSX Vector Test Least-Significant Bit by Byte XX2-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 28 of 30)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Compliancy Subsets</th>
<th>Linux Optional Category</th>
<th>Always Optional Category</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>956</td>
<td>VSX Vector Test for software Square Root Double-Precision XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>956</td>
<td>VSX Vector Test for software Square Root Single-Precision XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>957</td>
<td>VSX Vector Test Data Class Double-Precision XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>958</td>
<td>VSX Vector Test Data Class Single-Precision XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>960</td>
<td>VSX Vector Extract Exponent Double-Precision XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>960</td>
<td>VSX Vector Extract Exponent Single-Precision XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>961</td>
<td>VSX Vector Extract Significand Double-Precision XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>961</td>
<td>VSX Vector Extract Significand Single-Precision XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>962</td>
<td>VSX Vector Blend Variable Byte 8RR:XX4-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>963</td>
<td>VSX Vector Blend Variable Doubleword 8RR:XX4-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>963</td>
<td>VSX Vector Blend Variable Word 8RR:XX4-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>964</td>
<td>VSX Vector Byte-Reverse Doubleword XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>965</td>
<td>VSX Vector Byte-Reverse Halfword XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>966</td>
<td>VSX Vector Byte-Reverse Quadword XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>967</td>
<td>VSX Vector Byte-Reverse Word XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>967</td>
<td>VSX Vector Evaluate 8RR:XX4-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>969</td>
<td>VSX Vector Extract Unsigned Word XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>970</td>
<td>VSX Vector Generate PCV from Byte Mask X-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>971</td>
<td>VSX Vector Generate PCV from Doubleword Mask X-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>972</td>
<td>VSX Vector Generate PCV from Halfword Mask X-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>973</td>
<td>VSX Vector Generate PCV from Word Mask X-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>969</td>
<td>VSX Vector Insert Word XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>976</td>
<td>VSX Vector Logical AND XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>976</td>
<td>VSX Vector Logical Equivalence XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>979</td>
<td>VSX Vector Logical NAND XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>980</td>
<td>VSX Vector Logical NOR XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>981</td>
<td>VSX Vector Logical OR XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>981</td>
<td>VSX Vector Logical OR with Complement XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>981</td>
<td>VSX Vector Logical XOR XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>981</td>
<td>VSX Vector Logical XOR with Complement XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>981</td>
<td>VSX Vector Logical XOR XX3-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>981</td>
<td>VSX Vector Logical XOR with Complement XX3-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>983</td>
<td>VSX Move From Accumulator X-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>982</td>
<td>VSX Vector Merge High Word XX3-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>982</td>
<td>VSX Vector Merge Low Word XX3-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>984</td>
<td>VSX Move To Accumulator X-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>985</td>
<td>VSX Vector Permute XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>986</td>
<td>VSX Vector Permute Doubleword Immediate XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>987</td>
<td>VSX Vector Permute Right-indexed XX2-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>987</td>
<td>VSX Vector Permute Extended 8RR:XX4-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>988</td>
<td>VSX Vector Select XX4-form</td>
</tr>
<tr>
<td>12110 ...</td>
<td>MMA</td>
<td></td>
<td></td>
<td></td>
<td>xxvtxf</td>
<td>v3.1</td>
<td></td>
<td></td>
<td>989</td>
<td>VSX Set Accumulator to Zero X-form</td>
</tr>
</tbody>
</table>

Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 29 of 30)
### Figure 94. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 30 of 30)

1. **Instruction**
   - `/` Instruction bit that corresponds to a reserved field, must have a value of 0, otherwise invalid form.
   - `. ` Instruction bit that corresponds to an operand bit, may have a value of either 0 or 1.
   - `0` Instruction bit that corresponds to an opcode bit having a value 0.
   - `1` Instruction bit that corresponds to an opcode bit having a value 1.

2. **OpenPOWER Compliancy Subsets**
   - `X...` Instruction included in the Scalar Fixed-Point Compliancy subset
   - `.X...` Instruction included in the Scalar Fixed-Point + Floating-Point Compliancy subset.
   - `.X...` Instruction included in the Linux Compliancy subset.
   - `...X` Instruction included in the AIX Compliancy subset.

3. **Linux Optional Category**
   - AMO Instruction part of Atomic Memory Operations category.
   - BFP128 Instruction part of Quad-Precision Floating-Point category.
   - BHRB Instruction part of Branch History Rolling Buffer category.
   - DFP Instruction part of Decimal Floating-Point category.
   - EBB Instruction part of Event-Based Branch category.
   - MMA Instruction part of Matrix-Multiplication Assist category.

4. **Always Optional Category**
   - MMA Instruction part of Matrix-Multiplication Assist category.

5. **Version**
   - P1 Instruction introduced in POWER Architecture.
   - P2 Instruction introduced in POWER2 Architecture.
   - PPC Instruction introduced in PowerPC Architecture prior to v2.00.
   - v2.00 Instruction introduced in PowerPC Architecture Version 2.00.
   - v2.01 Instruction introduced in PowerPC Architecture Version 2.01.
   - v2.02 Instruction introduced in PowerPC Architecture Version 2.02.
   - v2.03 Instruction introduced in Power ISA Version 2.03.
   - v2.04 Instruction introduced in Power ISA Version 2.04.
   - v2.05 Instruction introduced in Power ISA Version 2.05.
   - v2.06 Instruction introduced in Power ISA Version 2.06.
   - v2.07 Instruction introduced in Power ISA Version 2.07.
   - v3.0 Instruction introduced in Power ISA Version 3.0.
   - v3.0B Instruction introduced in Power ISA Version 3.0B.
   - v3.0C Instruction introduced in Power ISA Version 3.0C.
   - v3.1 Instruction introduced in Power ISA Version 3.1.

6. **Privilege**
   - `P` Denotes an instruction that is treated as privileged.
   - `O` Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor-privileged for mtspr), depending on the SPR or PMR number.
   - `PI` Denotes an instruction that is illegal in privileged state.
   - `HV` Denotes an instruction that can be executed only in hypervisor state.
   - `UV` Denotes an instruction that can be executed only in ultravisor state.
7. **Mode Dependency.**

Except as described below and in Section 1.10.3, "Effective Address Calculation", in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

- **CT** If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
- **SR** The setting of status registers (such as XER and CR0) is mode-dependent.
- **SF=1** The instruction can be executed only in 64-bit mode.